Thesis for The Degree of Doctor of Philosophy

Visual GUI Testing:
Automating High-Level Software Testing in Industrial Practice

Emil Alégroth

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden, 2013

Visual GUI Testing:
Automating High-Level Software Testing in
Industrial Practice

EMIL ALÉGROTH

Copyright © 2015 Emil Alégroth
except where otherwise stated.
All rights reserved.

Technical Report No 117D
ISSN XXXX-YYY
ISBN XXXXXX
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Göteborg, Sweden 2015.

To Therese, Alexandra and my supporting family

Abstract

Software Engineering is at the verge of a new era where continuous releases are becoming more common than planned long-term projects. In this context, test automation will become essential on all levels of system abstraction to meet the market's demands of speedy delivery and high software quality. Hence, automated tests are required from low-level software components, tested with unit tests, up to the pictorial graphical user interface (GUI), tested with user emulated system and acceptance tests. Thus far, research has provided industry with a plethora of automation solutions for lower level testing but GUI level testing is still primarily a manual, and therefore costly and tedious, activity in practice.

We have identified three generations of GUI-based automated testing. The first (1st) generation relied on recorded, exact, GUI coordinates but was soon discarded due to unfeasible maintenance costs caused by fragility to GUI change. Second (2nd) generation tools, also referred to as component-, widget- or tag-based tools instead operate against some model of the GUI by hooking into the system's GUI architecture, libraries or application programming interfaces (APIs). Whilst tools of this approach are extensively used in practice, and are suitable to test system behavior, they are inflexible by being restricted to specific GUI technologies, programming languages and platforms.

The third (3rd) generation, referred to as Visual GUI Testing (VGT), is an emerging technique in industrial practice with properties that mitigate the challenges experienced with previous generations of GUI testing. VGT is defined as a tool-driven automated test technique where image recognition is used to interact with, and assert, a system's behavior through its pictorial GUI as it is shown to the user in user-emulated system or acceptance tests. VGT thereby provides test results of quality on par with a human tester and is therefore an effective complement to reduce also the aforementioned challenges of manual testing in practice. However, despite these benefits, the technique is only sparsely used in industrial practice and the academic body of knowledge holds only limited empirical support for the technique's industrial viability.

This thesis presents a holistic evaluation of VGT's capabilities in industrial practice, obtained through a series of case studies and experiments performed in academia and Swedish industry. The research follows an incremental methodology that began with academic experimentation with VGT tools, followed by studies in industrial practice that was concluded with a case study in a company that had used VGT for several years. Results of the research show that VGT is a viable technique for use in industrial practice with better defect-finding ability than manual tests, ability to test any GUI based system, high learnability, feasible maintenance costs and both short and long-term company benefits. However, there are still challenges, problems and limitations that must be considered for the successful adoption, use and long-term use of VGT in a company, the most crucial being that suitable development and maintenance practices are used. This thesis thereby concludes that VGT can be used in industrial practice and also provides guidance to practitioners that seek to do so. Additionally, this work provides a stepping stone for academia to explore new test solutions that build on image recognition technology.

Keywords

Software Engineering, Automated Testing, Visual GUI Testing, Industrial Research, Empirical Research, Applicability and Feasibility

Acknowledgments

First and foremost, my deepest thanks go to my main supervisor, friend and mentor Professor Robert Feldt whose belief in me and unwavering support made this thesis possible. We have had an amazing journey together and you have not just taught me how to be a researcher but a better person as well, something that I will cherish forever.

Second, my thanks go to my second supervisor, Associate professor Helena Holmström-Olsson, whose positive attitude, support and advice have been a great source of inspiration and help, both in times of joy and despair.

Next I want to thank my examiner Professor Gerardo Scheider and all my past and present colleagues at the Software Engineering division at Chalmers University of Technology whose guidance and support has been invaluable for the completion of my thesis work. In particular I would like to thank Dr. Ana Magazinius, Dr. Ali Shahrokni, Dr. Joakim Pernstål, Pariya Kashfi, Antonio Martini, Per Lenberg, Associate professor Richard Berntsson Svensson, Professor Richard Torkar and Professor Jan Bosch for many great experiences but also for always being there to listen to and support my sometimes crazy ideas. Additionally, I want to thank Bogdan Marculescu and Professor Tony Gorschek who, together with Robert, convinced me, in their own way, to proceed a PhD. Further, I want to thank my international research collaborators, in particular Professor Atif Memon, Rafael Oliveira and Zebao Gao who made a research visit in the US a wonderful experience.

However, this thesis had not been completed without the support of my loving wife, and mother of my wonderful Alexandra, Therese Alégroth. You have been my rock and the person I could always rely on when times were tough. Thanks also go to my mother Anette, father Tomas and sister Mathilda for believing in me and for their sacrifices to ensure that I could pursue this dream. Further, I want to thank my friends for always being there and I hope that one day, perhaps after reading my thesis, that you will understand what I do for a living.

I also want to thank my industrial collaborators, in particular the staff at Saab AB, Michel Nass, the staff at Inceptive, Geoffrey Bache, the Software Center and everyone else that has helped, supported and believed in my research.

This research has been conducted in a joint research project financed by the Swedish Governmental Agency of Innovation Systems (Vinnova), Chalmers University of Technology and Saab AB. My studies were also supported by the Swedish National Research School for Verification and Validation (SWELL), funded by Vinnova.

List of Publications

Appended papers

This thesis is primarily supported by the following papers:

1. E. Börjesson, R. Feldt, "Automated System Testing using Visual GUI Testing Tools: A Comparative Study in Industry"
   Proceedings of the 5th International Conference on Software Testing Verification and Validation (ICST'2012), Montreal, Canada, April 17-21, 2013 pp. 350-359.

2. E. Alégroth, R. Feldt, H. H. Olsson, "Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study"
   Proceedings of the 6th International Conference on Software Testing Verification and Validation (ICST'2013), Luxenbourg, March 18-22, 2013.

3. E. Alégroth, R. Feldt, L. Ryrholm, "Visual GUI Testing in Practice: Challenges, Problems and Limitations"
   Published in the Empirical Software Engineering Journal, 2014.

4. E. Alégroth, R. Feldt, P. Kolström, "Maintenance of Automated Test Suites in Industry: An Empirical study on Visual GUI Testing"
   In submission.

5. E. Alégroth, R. Feldt, "On the Long-term Use of Visual GUI Testing in Industrial Practice: A Case Study"
   In submission.

6. E. Alégroth, G. Zebao, R. Oliviera, A. Memon, "Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: An Empirical Study"
   Proceedings of the 8th International Conference on Software Testing Verification and Validation (ICST'2015), Graz, Austria, April 13-17, 2015

7. E. Alégroth, J. Gustafsson, H. Ivarsson, R. Feldt, "Replicating Rare Software Failures with Visual GUI Testing: An Industrial Success Story"
   In submission.

Other papers

The following papers are published but not appended to this thesis, either due to overlapping contents to the appended papers, contents not related to the thesis or because the contents are of less priority for the thesis main conclusions.

1. E. Börjesson, R. Feldt, "Structuring Software Engineering Case Studies to Cover Multiple Perspectives"
   Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Miami Beach, Florida, USA, July 1-3, 2011.

2. E. Alégroth, M. Nass, H. H. Olsson, "JAutomate: a Tool for System- and Acceptance-test Automation"
   Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST'2013), Luxenbourg, March 18-22, 2013.

3. E. Alégroth, "Random Visual GUI Testing: Proof of Concept"
   Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2013), Boston, Massachusetts, USAs, June 27-29, 2013.

4. G. Liebel, E. Algroth and R.Feldt, "State-of-Practice in GUI-based System and Acceptance Testing: An Industrial Multiple-Case Study"
   Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), 2013.

5. E. Algroth and R.Feldt, "Industrial Application of Visual GUI Testing: Lessons Learned"
   Chapter of the book Continuous Software Engineering published by Springer, 2014.

6. E. Alégroth, G. Bache, E. Bache, "On the Industrial Applicability of TextTest: An Empirical Case Study"
   Proceedings of the 8th International Conference on Software Testing Verification and Validation (ICST'2015), Graz, April 13-17, 2015

7. R. Oliviera, E. Alégroth, G. Zebao, A. Memon, "Definition and Evaluation of Mutation Operators for GUI-level Mutation Analysis"
   Proceedings of the 10th Mutation Workshop (Mutation'2015), Graz, Austria, April 13, 2015

Statement of contribution

In all listed papers, the first author was the primary contributor to the research idea, design, data collection, analysis and/or reporting of the research work.

Contents

Abstract **v**

Acknowledgments **vii**

List of Publications **ix**

1 Introduction **1**
1.1 Introduction 1
1.2 Software engineering and the need for testing 3
1.2.1 Software Testing 3
1.2.2 Automated Software Testing 5
1.2.3 Automated GUI-based Software Testing 7
1.2.3.1 1st generation: Coordinate-based 7
1.2.3.2 2nd generation: Component/Widget-based . . 7
1.2.3.3 3rd generation: Visual GUI Testing 9
1.2.3.4 Comparison 11
1.3 Research problem and methodology 11
1.3.1 Problem background and motivation for research 13
1.3.2 Thesis research process 16
1.3.3 Research methodology 17
1.3.4 Case studies 18
1.3.4.1 Interviews 20
1.3.4.2 Workshops 21
1.3.4.3 Other 22
1.3.5 Experiments 24
1.3.6 Data analysis 24
1.4 Overview of publications 27
1.4.1 Paper A: Static evaluation 27
1.4.2 Paper B: Dynamic evaluation 28
1.4.3 Paper C: Challenges, problems and limitations 30
1.4.4 Paper D: Maintenance and return on investment 31
1.4.5 Paper E: Long-term use 33
1.4.6 Paper F: VGT-GUITAR 35
1.4.7 Paper G: Failure replication 37
1.5 Contributions, implications and limitations 38
1.5.1 Applicability of Visual GUI Testing in practice 39
1.5.2 Feasibility of Visual GUI Testing in practice 42

1.5.3 Challenges, problems and limitations with Visual GUI Testing in practice . . . 47
1.5.4 Solutions to advance Visual GUI Testing . . . 47
1.5.5 Implications . . . 48
1.5.5.1 Implications for practice . . . 48
1.5.5.2 Future research . . . 49
1.5.6 Threats and limitations of this research . . . 51
1.5.6.1 Internal validity . . . 51
1.5.6.2 External validity . . . 52
1.5.6.3 Construct validity . . . 52
1.5.6.4 Reliability/conclusion validity . . . 53
1.6 Thesis summary . . . 53

2 Paper A: Static evaluation 55
2.1 Introduction . . . 56
2.2 Related Work . . . 57
2.3 Case Study Description . . . 59
2.3.1 Pre-study . . . 60
2.3.2 Industrial Study . . . 62
2.4 Results . . . 64
2.4.1 Results of the Pre-study . . . 64
2.4.2 Results of the industrial study . . . 68
2.5 Discussion . . . 71
2.6 Conclusion . . . 73

3 Paper B: Dynamic evaluation 75
3.1 Introduction . . . 76
3.2 Related Work . . . 77
3.3 Research methodology . . . 78
3.3.1 Research site . . . 79
3.3.2 Research process . . . 80
3.4 Results and Analysis . . . 81
3.4.1 Pre-transition . . . 81
3.4.2 During transition . . . 83
3.4.2.1 VGT test suite maintenance for improvement . 85
3.4.2.2 VGT test suite maintenance required due to SUT change . . . 86
3.4.3 Post-transition . . . 88
3.5 Discussion . . . 91
3.5.1 Threats to validity . . . 94
3.6 Conclusion . . . 94

4 Paper C: Challenges, problems and limitations 97
4.1 Introduction . . . 98
4.2 Background and Related work . . . 100
4.3 Industrial case study . . . 102
4.3.1 The industrial projects . . . 103
4.3.2 Detailed data collection in Case 1 . . . 105
4.3.3 Detailed data collection in Case 2 . . . 107

4.3.4 The VGT suite 108
4.4 Results and Analysis 110
4.4.1 Test system related CPLs 111
4.4.1.1 Test system version 112
4.4.1.2 Test system (General) 115
4.4.1.3 Test system (Defects) 117
4.4.1.4 Test company specific CPLs 118
4.4.1.5 Test system (Environment) 119
4.4.2 Test tool related CPLs 119
4.4.2.1 Test tool (Sikuli) related CPLs 119
4.4.2.2 Test application 124
4.4.3 Support software related CPLs 125
4.4.4 CPL Summary 127
4.4.5 Potential CPL solutions 129
4.4.6 Defect finding ability, development cost and return on investment (ROI) 131
4.5 Discussion 138
4.5.1 Challenges, Problems, Limitations and Solutions 138
4.5.2 Defects and performance 140
4.5.3 Threats to validity 142
4.6 Conclusions 143

5 Paper D: Maintenance and return on investment 145
5.1 Introduction 146
5.2 Related work 147
5.3 Methodology 148
5.3.1 Phase 1: Interview study 149
5.3.2 Phase 2: Case study Setting 150
5.3.3 Phase 2: Case study Procedure 153
5.4 Results and Analysis 155
5.4.1 Quantitative results 156
5.4.1.1 Modeling the cost 159
5.4.2 Qualitative results 161
5.4.2.1 Phase 1: Interview results 161
5.4.2.2 Phase 2: Observations 163
5.4.2.3 Phase 2: Factors that affect the maintenance of VGT scripts 164
5.5 Discussion 168
5.5.1 Threats to validity 170
5.6 Conclusions 171

6 Paper E: Long-term use 173
6.1 Introduction 174
6.2 Related work 176
6.3 Methodology 177
6.3.1 Case company: CompanyX 177
6.3.2 Research design 179
6.4 Results and Analysis 184
6.4.1 Results for RQ1: VGT adoption 184

6.4.2 Results for RQ2: VGT benefits 185
6.4.3 Results for RQ3: VGT challenges 187
6.4.4 Results for RQ4: VGT alternatives 189
6.4.5 Quantification of the Qualitative Results 192
6.5 Guidelines for adoption and use of VGT in industrial practice . 194
6.5.1 Adoption of VGT in practice 196
6.5.2 Use of VGT in practice 197
6.5.3 Long-term use of VGT in practice 198
6.6 Discussion 199
6.6.1 Threats to Validity 201
6.7 Conclusions 202
6.8 Appendix A: Interview Questions 203

7 Paper F: VGT-GUITAR 205
7.1 Introduction 206
7.2 Background and Motivation 207
7.3 Methodology 209
7.3.1 Experiment: Fault detection and False results 209
7.3.2 Case study: Applicability in practice 213
7.4 Results and Analysis 214
7.4.1 Experiment 214
7.4.2 Case study 216
7.5 Discussion 220
7.5.1 Threats to Validity 221
7.6 Related Work 222
7.7 Conclusions 223

8 Paper G: Failure replication 225
8.1 Failure replication 226
8.2 Success story acquisition 226
8.3 Success story at Saab 227
8.4 Discussion 230
8.5 Lessons learnt 232

Bibliography 233

Chapter 1

Introduction

1.1 Introduction

Today, software is ubiquitous in all types of user products, from software applications to cars, mobile applications, medical systems, etc. Software allows development organizations to broaden the number of features in their products, improve the quality of these features and provide customers with post-deployment updates and improvements. In addition, software has shortened the time-to-market in many product domains, a trend driven by the market need for new products, features and higher quality software.

However, these trends place new time constraints on software development organizations that limit the amount of requirements engineering, development and testing that can be performed on new software [1]. For testing, these time constraints imply that developers can no longer verify and validate the software's quality with manual test practices since manual testing is associated with properties such as high cost, tediousness and therefore error-proneness [2–7]. These properties are a particular challenge in the context of changing requirements where the tests continuously need to be rerun for regression testing [8, 9].

Automated testing has been suggested as the solution to this challenge since automation allows tests to be run more frequently and at lower cost [4, 7, 10]. However, most automated test techniques have prerequisites that prohibit their use on software written in certain programming languages, for certain operating systems, platforms, etc. [4, 11–13]. Additionally, most automated test techniques operate on a lower level of system abstraction, i.e. against the backend of the system. One such, commonly used, low-level test technique is automated unit testing [14]. Whilst unit tests are applicable to find defects in individual software components, its use for system and acceptance testing is still a subject of ongoing debate [15, 16]. Test techniques exist for automated system and acceptance testing that interact with the system under test (SUT) through hooks into the SUT or its GUI. However, these techniques do not verify that the pictorial GUI, as shown to the user, behaves or appears correctly. These techniques therefore have limited ability to fully automate manual, scenario-based, regression test cases, in the continuation of this thesis referred to as *manual test cases*. Consequently, industry is in need of a

flexible and GUI-based test automation technique that can emulate human tester behavior to mitigate the challenges associated with current manual and automated test techniques.

In this thesis we introduce and evaluate Visual GUI Testing (VGT). VGT is a term we have defined that encapsulates all tools that use image recognition to interact with a SUT's functionality through the bitmaps shown on the SUT's pictorial GUI. These interactions are performed with user emulated keyboard and mouse events that make VGT applicable on almost any GUI-driven application and to automate test cases that previously had to be performed manually. Consequently, VGT has the properties that software industry is looking for in a flexible, GUI-based, automated test technique since the technique's only prerequisite is that a SUT has a GUI. A prerequisite that only limits the technique's applicability and usefulness for, for instance, server or other backend software.

However, at the start of this thesis work, the body of knowledge on VGT was limited to analytical research results [17] regarding VGT tools, i.e. Triggers [18], VisMap [19] and Sikuli [20]. Hence, no empirical evidence existed regarding the technique's applicability or feasibility of use in industrial practice. Applicability that, in this thesis, refers to factors such as a test technique's defect-finding ability, usability for regression, system and acceptance testing, learnability and flexibility of use for different types of GUI-based software. Feasibility, in turn, refers to the long-term applicability of a technique, including feasible development and maintenance costs, usability under strict time constraints and suitable time until the technique provides positive return on investment (ROI). Empirical evidence on these factors are key to understand the real life complexities of using the technique, to build best practices and to advance its use in industrial practice [17, 21]. However, such evidence can only be acquired through an incremental process that evaluates the technique from several perspectives and different industrial contexts. This thesis work was therefore performed in Swedish software industry, with different projects, VGT tools and research techniques to fulfill the thesis research objective. Hence, to acquire evidence for, or against, the applicability and feasibility of adoption, use and viability of VGT in industrial practice, including what challenges, problems and limitations that are associated with these activities. Work that consequently resulted in an overall understanding of the current state-of-practice of VGT, what impedes its continued adoption and a final, yet positive, conclusion regarding the long-term viability of VGT in industrial use.

The results presented in this introductory chapter (Chapter 1) are structured as follows. First, an introduction is given in Section 1.1 followed by a background to this research, including; manual, automated and automated GUI-based testing. Section 1.3 then presents the research problem, questions and the methodology. This section also details the different research methods that were used and how the included papers contribute to answer the thesis research questions. An overview, and summaries, of the included papers are then given in Section 1.4. Section 1.5 then presents the syntheses of included papers and finally the thesis introduction is concluded in a summary in Section 1.6.

1.2 Software engineering and the need for testing

Software engineering is the application of engineering best practices in a structured process to design, develop and maintain software of high quality [22]. Several software development processes have been defined such as plan-driven, incremental and agile development processes [23, 24]. These processes can be divided into three fundamental activities: requirements engineering, development (design and implementation) and verification and validation.

Requirements engineering refers to the activity of elicitation, specification and modeling of the software's requirements, i.e. the needs of the customer/user. Hence, features, functions and qualities that the developed software must include [25, 26]. In turn, development is the activity of designing and realizing the requirements in software that fulfills the user's needs. Finally, verification and validation, traditionally, is the activity of evaluating that the developed software conforms to the requirements [1], most commonly achieved through testing.

Tests for verification and validation are therefore a tightly coupled counterpart to requirements [27]. Hence, whilst the quality of a software system is determined by how well each process activity is performed, it is through testing that this quality is measured. Measurements that can be taken throughout the development process, i.e. early with reviews of documents or code or late with customer acceptance tests. Testing is therefore an essential activity in all software engineering, regardless of process or development objective.

1.2.1 Software Testing

Software testing for verification and validation is a core, but also costly, activity that can make up for 20-50 percent of the cost of a software development project [1, 28, 29]. Verification is defined as the practice of assuring that the SUT conforms to its requirements, whilst validation is defined as the practice of assuring that the SUT conforms to the requirements and fulfills the user's needs [25, 26].

System view	System layers	System components	Manual testing		Automated testing
Front-end	Pictorial GUI	Bitmaps	Regression system and acceptance testing	Exploratory testing	Visual GUI Testing
	GUI model	**Hooks into:** GUI API/Toolkit (GUI) Source code/ architecture			Component/Widget/ Tag-based GUI-testing
Back-end	System core	SW architecture Technical interfaces SW components Classes Functions/methods	Reviews, unit testing and integration testing		Unit-testing and integration testing

Figure 1.1: *Theoretical, layered, model of a System and the manual/automated techniques generally used to test the different layers.*

Testing for the purpose of verification can be split into three types; unit, integration and system testing [30], which are performed on different levels of system abstraction [16, 26, 31] as shown in Figure 1.1. A unit test verifies that the behavior of a single software component conforms to its low-level functional requirement(s) and is performed either through code reviews or more commonly through automated unit tests [9, 11, 14, 15, 32–34]. In turn, integration tests verify the conformance of several components' interoperability between each other and across layers of the SUT's implementation [16, 30]. Components can in this context be single methods or classes but also hardware components in embedded systems. Finally, system tests are, usually, scenario-based manual or automated tests that are performed either against the SUT's technical interfaces or the SUT's GUI to verify that the SUT, as a whole [30], conforms to its feature requirements [35–37]. However, scenario-based tests are also used to validate the conformance of a SUT in acceptance tests that are performed either by, or with, the SUT's user or customer [35–38]. The key difference between system and acceptance test scenarios is therefore how representative they are of the SUT's real-world use, i.e. the amount of domain knowledge that is embedded in the test scenario.

Testing is also used to verify that a SUT's behavior still conforms to the requirements after changes to the SUT, i.e. regression tests. Regression tests can be performed with unit, integration, system or acceptance test cases that have predefined inputs for which there are known, expected, outputs [9]. Inputs and outputs that are used to stimulate and assert various states of the SUT. As such, the efficiency of a regression test suite is determined by the tests' coverage of the SUT's components, features, functions, etc [34, 39], i.e. the amount of a SUT's states that are stimulated during test execution. This also limits regression tests to finding defects in states that are explicitly asserted, which implies that the test coverage should be as high as possible. However, for manual regression tests, high coverage is costly, tedious and error-prone [2–7], which is the primary motivation why automated testing is needed and should be used on as many different levels of system abstraction as possible [16, 40]. Especially in the current market where the time available for testing is shrinking due to the demands for faster software delivery [1]. Demands that have transformed automated testing from "want" to a "must" in most domains.

However, whilst lower levels of system abstraction are well supported by automated regression test techniques, tools and frameworks, there is a lack of automated techniques for testing through the pictorial GUI, i.e. the highest level of system abstraction. Thus, a lack of support that presents the key motivator for the research presented in this thesis.

To cover any lack of regression test coverage, exploratory testing, defined as simultaneous learning, test design and test execution, is commonly used in industrial practice [41, 42]. The output of exploratory testing is a defect but also the scenario(s) that caused the defect to manifest, i.e. scenarios that can be turned into new regression tests. This technique has been found to be effective [43] but has also been criticized for not being systematic enough for fault replication. Further, the practice requires decision making to guide the testing and is therefore primarily performed manually, despite the existence of a few automated exploratory testing tools, e.g. CrawlMan [44]. However, automated exploratory testing is still an unexplored research area that war-

rants more research, including automated GUI-based exploratory testing since it could help mitigate the challenges associated with manual verification and validation, e.g. cost.

In summary, testing is used in industrial practice on different levels of system abstraction for verification and validation of a SUT's conformance to its requirements. However, much of this testing is manual, which is costly, tedious and error prone, especially for manual regression testing, which is suggested as solvable with automated testing. More research is therefore warranted into new automated test techniques and in particular techniques that operate against the SUT's highest level of system abstraction, i.e. the pictorial GUI.

1.2.2 Automated Software Testing

There are two key motivators for the use of automated testing in industrial practice; (1) to improve software quality and (2) to lower test related costs [40].

***Software quality*:** Automated tests help raise software quality through higher execution speed than manual tests that allow them to be executed more frequently [16, 40]. Higher test frequency provides faster feedback to the developers regarding the quality of the software and enables defects to be caught and resolved earlier. In turn, quick defect resolution lowers the project's development time and mitigates the chance of defect propagation into customer deliveries. Early defect detection also mitigates synergy effects to occur between defects, for instance that two or more defects cause a joint failure which root-cause therefore becomes more difficult and costly to find.

However, a prerequisite for any automated test technique to be used frequently is that the tests have reasonable test execution time. This prerequisite is particularly important in contexts where the tests are used for continuous integration, development and deployment [45]. Hence, contexts where the test suites should be executed each time new code is integrated to the SUT, e.g. on commit, which cause the tests to set the pace for the highest possible frequency of integration. This pacing is one reason why automated unit tests [9, 11, 14, 15, 32–34] are popular in industrial practice since several hundred unit tests can be executed in a matter of minutes. In addition, unit tests are popular in agile software development companies, where they are used to counteract regression defects [46] caused by change or refactoring that is promoted by the process [47, 48].

***Lower cost*:** Automated testing is also used to lower the costs of testing by automating tests, or parts of tests, that are otherwise performed manually. However, there are still several costs associated with automated tests that need to be considered.

First, all automated test techniques require some type of tool that either needs to be acquired, bought and/or developed. Next, the intended users of the tool need be given training or time to acquire knowledge and experience with the tool and its technique before it can be used. Knowledge and experience that might be more or less cumbersome to acquire dependent on the technique's complexity [40]. This complexity implies that techniques with high learnability are more favorable from a cost perspective since they require less training.

Furthermore, adoption of test automation is associated with organizational changes, e.g. new or changed roles, which adds additional costs, especially if

the organizational changes affect the company's processes, e.g. due to changes of the intended users' responsibilities. Additionally, many automated test techniques have prerequisites that prohibit their use to certain systems written in specific programming languages, operating systems and platforms [4,11–13]. Therefore it is necessary to perform a pilot project to (1) evaluate if the new technique is at all applicable for the intended SUT and (2) for what types of tests the technique can be used. Thus a pilot project is an important activity but also associated with a, sometimes substantial, cost. However, several of these costs are often overlooked in practice and are thereby "hidden" costs associated with any change to a software process.

Second, for established systems, and particularly legacy systems, a considerable cost of adopting a new test technique is associated with the development of a suitably large test suite that provides test coverage of the SUT. Hence, since automated testing is primarily used for regression testing, test coverage, as stated in Section 1.2.1, is required for the testing to be efficient and valuable in finding defects.

However, this brings us to the ***third*** cost associated with automated testing which is maintenance of test scripts. Maintenance constitutes a continuous cost for all automated testing that grows with the size of the test suite. This maintenance is required to keep the test scripts aligned with the SUT's requirements [49], or at least its behavior, to ensure that test failures are caused by defects in the SUT rather than intended changes to the SUT itself, i.e. failures referred to as false positives. However, larger changes to the SUT can occur and the resulting maintenance costs can, in a worst case, become unreasonable [12]. These costs can however be mitigated through engineering best practices, e.g. modular test design [16, 40, 50]. However, best practices takes time to acquire, for any technique, and are therefore often missing, also for VGT.

Hence, these three costs must be compared together to the value provided by the automated tests, for instance value in terms of defects found or to the costs compared to alternative test techniques, e.g. manual testing. The reason for the comparison is to identify the point in time when the costs of automation break even with the alternatives, i.e. when return on investment (ROI) is achieved. Hence, for any automated test technique to be feasible, the adoption, development and maintenance costs must provide ROI and it should do so as quickly as possible. Consequently, an overall view of costs, value and other factors, e.g. learnability, adoptability and usability, is required to provide an answer if a test automation technique is applicable and feasible in practice. These factors were therefore evaluated during the thesis work to provide industrial practitioners with decision support of when, how and why to adopt and use VGT.

In summary, automated testing helps improve SUT quality and lower project costs [40]. However, the costs of automated testing can still be substantial and must therefore be evaluated against other alternative techniques to identify when and if the adoption of a new technique provides positive ROI.

1.2.3 Automated GUI-based Software Testing

Automated software testing has several benefits over manual testing, e.g. improved test frequency, but there are also challenges, for instance, that most techniques operate on a lower level of system abstraction. However, there is a set of automated test techniques that operate against, or through, the SUT's GUI that can be used for higher level testing. To clarify the differences between these types of GUI-based testing techniques we have divided them into three chronologically defined generations [51]. The difference between each generation is how they interact with the SUT, i.e. with exact coordinates, through hooks into the SUT's GUI or image recognition. The following section presents key properties of the three generations to provide the reader with contextual information for the continuation of the thesis.

1.2.3.1 1st generation: Coordinate-based

1st generation GUI-based test automation uses exact coordinates on the screen to interact with the SUT [3]. These coordinates are acquired by recording manual interaction with the SUT and are then saved to scripts that can be replayed for automated regression testing, which improves test frequency. However, the technique is fragile, even minor changes to a GUI's layout can cause an entire test suite to fail, resulting in frequent and costly maintenance [3,52,53]. Therefore, the technique has mostly been abandoned in practice but is commonly integrated as one basic component into other test automation frameworks and tools, e.g. JUnit [53] and Sikuli [54]. However, because of the technique's limited stand-alone use in practice it will not be discussed to any extent in this thesis.

1.2.3.2 2nd generation: Component/Widget-based

2nd generation GUI-based testing tools stimulate and assert the SUT through direct access to the SUT's GUI components or widgets by hooks into the SUT, e.g. into its GUI libraries or toolkits [12]. Synonyms for this technique are Component-, Widget- or Tag-based GUI testing and is performed in industrial practice with tools such as Selenium [55], QTP [56], etc.

These tools can achieve robust test case execution, e.g. few false test results, due to the tools' access and tight coupling to the SUT's internal workings, e.g. GUI events and components' ID numbers, labels, etc. These GUI events can also be monitored in a few tools to automatically synchronize the test script with the SUT, which would otherwise require the user to manually specify synchronization points in the scripts, e.g. static delays or delays based on GUI state transitions. Synchronization is a common challenge for all GUI-based test techniques because the test scripts run asynchronously to the SUT.

Another advantage of SUT access is that some of these tools can improve test script execution time by forcing GUI state transitions and bypass cosmetic, timed, events such as load screens, etc.

Further, most 2nd generation tools support record and replay, which lowers test development costs. In addition, most tools support the user by managing GUI components' property data, e.g. ID numbers, labels, component types,

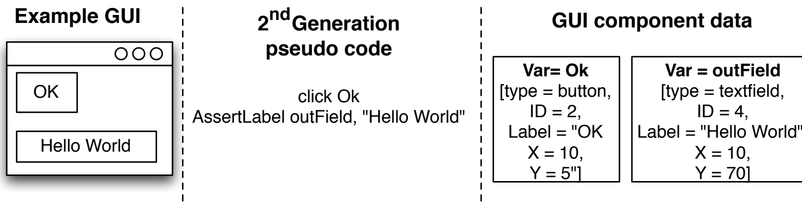

Figure 1.2: *Pseudocode example of a 2nd generation test script for a simple application where GUI components are identified through their properties (Tags), in this case, associated with a user defined variable.*

etc [57]. This functionality is required since these properties are unintuitive without technical or domain knowledge, e.g. an ID number or component type is not enough for a human to intuitively identify a component. However, combined, groups of properties allow the tester to distinguish between components, exemplified with pseudocode in Figure 1.2.

Some 2nd generation tools, e.g. GUITAR [58], also support GUI ripping that allow the tools to automatically extract GUI components, and their properties, from the SUT's GUI and create a model over possible interactions with the SUT. These models can then be traversed to generate scenarios of interactions that can be replayed as test cases, a technique typically referred to as model-based testing [59–63]. As such, provided that the interaction model contains all GUI components, it becomes theoretically possible to automatically achieve full feature coverage of the SUT since all possible scenarios of interactions can be generated. However, in practice this is not possible since the number of test cases grow exponentially with the number of GUI components and length of test cases that makes it unreasonable to execute all of them. This problem is referred to as the state-space explosion problem and is common to most model-based testing tools [59]. One way to mitigate the problem is to limit the number of interactions per generated test scenario but this practice also limits the tests' representativeness of real world use and stifles their ability to reach faulty SUT states.

Furthermore, because 2nd generation GUI-based tools' interact with the SUT through hooks into the GUI, these tests do not verify that the pictorial GUI conforms to the SUT's requirements, i.e. neither that its appearance is correct or that human interactions with it is possible. In addition, the tools require these hooks into the SUT to operate, which restricts their use to SUT's written in specific programming languages and for certain GUI libraries/toolkits. This requirement also limits the tools' use for testing of systems distributed over several physical computers, cloud based applications, etc., where the SUT's hooks are not accessible.

Another challenge is that the tools need to know what properties a GUI component has to stimulate and assert its behavior. Standard components, included in commonly used GUI libraries, e.g. JAVA Swing or AWT, are generally supported by most tools. However, for custom built components, e.g. user defined buttons, the user has to create custom interpreters or hooks for the tools to operate. However, these interpreters need to be maintained if the components are changed, which adds to the overall maintenance costs. Overall

maintenance costs that have been reported to, in some cases, be substantial in practice [10, 12, 16, 52].

However, there are also some types of GUI components that are difficult or can not be tested with this technique, e.g. components generated at runtime, since their properties are not known prior to execution of the system. As such, there are several challenges associated with 2nd generation GUI-based testing that limit the technique's flexibility of use in industrial practice.

In summary, 2nd generation GUI-based testing is associated with quick and often robust test execution due to their access to the SUT's inner workings. However, this access is a prerequisite for the technique's use that also limits its tools to test applications written is certain programming languages, with certain types of components, etc. As a consequence, the technique lacks flexibility in industrial use. Further, the technique does not operate on the same level of system abstraction as a human user and does therefore not verify that the SUT is correct from a pictorial GUI point of view, neither in terms of appearance or behavior. Additionally, the technique is associated with script maintenance costs that can be extensive and in worst cases infeasible [10, 12, 16, 52]. Consequently, 2nd generation GUI-based testing does not fully fulfill the industry's needs for a flexible and feasible test automation technique.

1.2.3.3 3rd generation: Visual GUI Testing

3rd generation GUI-based testing is also referred to as Visual GUI Testing (VGT) [64], and is defined as *a tool driven automated test technique where image recognition is used to interact with, and assert, a system's behavior through its pictorial GUI as it is shown to the user in user emulated system or acceptance tests.* The foundation for VGT was established in the early 90s by a tool called Triggers [18], later in the 90s accompanied by a tool called VisMap [19], which both supported image recognition based automation. However, at the time, lacking hardware support for the performance heavy image recognition algorithms made these tools unusable in practice [65]. Advances in hardware and image recognition algorithm technology have now mitigated this challenge [66] but it is still unknown if VGT, as a technique, is mature enough for industrial use. Thus providing one motivation the work presented in this thesis.

Several VGT tools are available in practice, both open source; Sikuli [20], and commercial; JAutomate [67], EggPlant [68] and Unified Functional Testing (UFT) [56], each with different benefits and drawbacks due to the tools' individual features [67]. However, common to all tools is that they use image recognition to drive scripts that allow them to be used on almost any GUI-driven application, regardless of implementation, operating system or even platform. As a consequence, VGT is associated with a high degree of flexibility. The technique does however only have limited usefulness for non-GUI systems, e.g. server-applications.

VGT scripts are written, or recorded, as scenarios that contain methods which are usually synonyms for human interactions with the SUT, e.g. mouse and keyboard events, and bitmap images. These images are used by the tools' image recognition algorithms to stimulate and assert the behavior of SUT through its pictorial GUI, i.e. in the same way as a human user. Consequently,

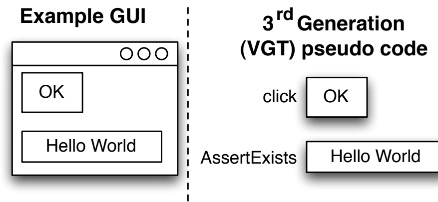

Figure 1.3: *Pseudocode example of 3rd generation (VGT) test case for a simple application. GUI components are associated with the application's GUI component images (bitmaps).*

VGT scripts are generally intuitive to understand, also for non-technical stakeholders, since the scripts' syntax is relatable to how the stakeholders would themselves interact with the SUT [20], e.g. click on a target represented by a bitmap and type a text represented by a string. This intuitiveness also provides VGT with high learnability also by technically awkward users [65].

A pseudo-code VGT script example is shown in Figure 1.3 that performs the same interactions as the example presented for 2nd generation GUI-based testing, presented in Figure 1.2, for comparison.

Conceptually, image recognition is performed in two steps during VGT script playback. First, the SUT's current GUI state is captured as a bitmap, e.g. in a screenshot of the computers desktop, which is sent together with the sought bitmap from the VGT script to the image recognition algorithm. Second, the image recognition algorithm searches for the sought bitmap in the screenshot and if it finds a match it returns the coordinates for the match that are then used to perform an interaction with the SUT's GUI. Alternatively, if the image recognition fails, a false boolean is returned or an exception is raised.

Different VGT tools use different algorithms but most algorithms rely on similarity-based matching which means that a match, i.e. sought bitmap, is found if it is within a percentile margin between the identified and sought bitmap image [20]. This margin is typically set to 70 to 80 percent of the original image to counteract failures due to small changes to a GUI's appearance, e.g. change of a GUI bitmap's color tint. However, similarity-based matching does not prevent image recognition failure when bitmaps are resized or changed completely.

Additionally, VGT scripts, similar to 1st and 2nd generation scripts, need to be synchronized with the SUT's execution. Synchronization in VGT is performed with built in functionality or methods that wait for a bitmap(s) to appear on the screen before the script can proceed. However, these methods also make VGT scripts slow since they cannot execute quicker than the state transitions of the GUI, which is a particular challenge for web-systems since waits also need to take network latency into account.

In summary, VGT is a flexible automated GUI-based test technique that uses tools with image recognition to interact and assert a SUT's behavior through its pictorial GUI. However, the technique's maturity is unknown and this thesis therefore aims to evaluate if VGT is applicable and feasible in industrial practice.

1.2.3.4 Comparison

To provide a general background and overview of the three generations of automated GUI-based testing, some of their key properties have been presented in Table 1.1. The table shows which properties that each technique has ("Y") or not ("N") or if a property is support by some, but not all, of the technique's tools ("S"). These properties were acquired during the thesis work as empirical results or through analysis of related work. However, they are not considered to be part of the thesis main contributions even though they support said contributions.

Several properties are shared by all techniques. For instance, they can all be used to automate manual test cases but only VGT tools also support bitmap assertions and user emulation and it is therefore the only technique that provides results of equal quality to manual tests. Further, all three techniques are perceived to support daily continuous integration and all techniques require the scripts to be synchronized with the SUT's execution. Finally, none of the techniques are perceived as replacements to manual testing since all of the techniques are designed for regression testing and therefore only find defects in system states that are explicitly asserted. In contrast, a human can use cognitive reasoning to determine if new, previously unexplored, states of the SUT are correct. Consequently, a human oracle [69] is required to judge if a script's outcome is correct or not.

Other properties of interest regard the technique's robustness to change. For instance, both 2nd and 3rd generation tools are robust to GUI layout change, assuming, for the 3rd generation, that the components are still shown on the screen after change. In contrast, 1st generation tools are fragile to this type of change since they are dependent on the GUI components' location being constant.

However, 1st generation tools, and also 3rd generation tools, are robust to changes to the SUT's GUI code whilst 2nd generation tools are not, especially if these changes are made to custom GUI components, the GUI libraries or GUI toolkits [12].

Finally, 1st and 2nd generation tools are robust to changes to the GUI components' bitmaps since none of the techniques care about the GUI's appearance. In contrast, 3rd generation tools fail if either the appearance or the behavior of the SUT is incorrect.

Consequently, the different techniques have different benefits and drawbacks that are perceived to make the techniques more or less applicable in different contexts.

1.3 Research problem and methodology

In this section, a summary of the background and the motivation for the research performed in this thesis work are presented. These are based on the challenges and gaps in knowledge and tooling presented in Sections 1.1 to 1.2.3.4. Additionally, the research objective is presented and broken down into four specific research questions that the thesis work aimed to answer through an incremental research process that is also presented. Finally, the research methodology and research methods used during the thesis work are discussed.

Property	**1st Gen.**	**2nd Gen.**	**3rd Gen.**
Independent of SUT platform	N	N	Y
Independent of SUT programming language	Y	S	Y
Non-intrusive test execution	N	S	Y
Emulates human user behavior	Y	N	Y
Open-source tool alternatives	Y	Y	Y
Supports manual test case automation	Y	Y	Y
Supports testing of custom GUI components	Y	S	Y
Supports bitmap-based assertions	N	S	Y
Supports testing of distributed systems	Y	S	Y
Supports daily continuous integration	Y	Y	Y
Robust to GUI layout change	N	Y	Y
Robust to system code change	Y	N	Y
Robust to bitmap GUI component change	Y	Y	N
Support script recording (as opposed to manual scripting)	Y	Y	S
Script execution time independent of SUT performance	N	N	N
Replacement of other manual/automatic test practices	N	N	N

Table 1.1: *The positive and negative properties of different GUI-based test techniques. All properties have been formulated such that a "Y" indicates that the property is supported by the technique. "N" indicates that the property is not supported by the technique. "S" indicates that some of the technique's tools supports the property, but most don't.*

1.3.1 Problem background and motivation for research

Background: Testing is the primary means by which companies verify and validate (V&V) their software. However, the costs of V&V ranges between 20-50 percent of the total costs associated with a software development project [1, 28, 29], which is a challenge that can be contributed to the extensive industrial use of manual, tedious, time consuming, and therefore error prone V&V practices [2–7]. Automated testing is generally proposed as the solution to this challenge, since automated test scripts execute systematically each time and with reduced human effort and cost [40]. However, this proposition presents new challenges for software development companies, such as *what automated testing do they need, how is it performed and how does it provide value?*

The most common type of automated testing in practice is automated unit testing [14, 33], which has been shown to be effective to find software defects. However, unit tests operate on a low level of system abstraction and they have therefore been debated to be ill suited for V&V of high level requirements [15, 16]. Automated unit testing therefore has a place in software development practice but should be complemented with test techniques also on higher levels of system abstraction to provide full automated coverage of the SUT [16]. For GUI-driven software this also includes automated testing of the pictorial GUI as shown to the user.

To acquire GUI automation coverage, many companies use 2nd generation GUI-based testing for automated system testing, for instance with the tool Selenium [55]. However, these tools interact with the SUT by hooking into its GUI libraries, toolkits or similar and therefore do not verify that human interaction with the SUT's pictorial GUI can be performed as expected [51]. Such verification requires an automated test technique that can operate on the same level of abstraction and with the same confidence and results as a human user.

In addition, most automated test techniques' are restricted to be used on SUTs that fulfill the tools' prerequisites, such as use of specific programming languages, platforms, interfaces for testing etc [4, 11–13]. These prerequisites are a particular challenge for legacy, or distributed, systems that are either not designed to support automated testing or lack the necessary interfaces for test automation. As a consequence, industry is in need of a flexible test automation technique with less, or easily fulfilled, prerequisites.

Further, the view that automated testing lowers test related cost is only partially true because test automation is still associated "hidden" costs and, in particular, maintenance costs [10, 12, 16, 40, 52]. Therefore, adoption of automated testing can lower the total development cost of a project by enabling faster feedback to developers that leads to faster defect resolution, but test related costs still remain or can even increase. As such, to fulfill industry's need for a flexible GUI-based test automation technique, a technique must be identified that is feasible long-term and which preferably provides quick ROI compared to manual testing. Such a technique must also provide value in terms of, at least, equal defect finding ability as manual testing and with low test execution time to facilitate frequent test execution.

Motivation: In theory, Visual GUI Testing (VGT) fulfills the industrial need for a flexible, GUI-based, automated test technique due to its unprece-

Paper	Objective	RQ1	RQ2	RQ3	RQ4
A	Static evaluation of VGT in practice	X	X	X	
B	Dynamic evaluation of VGT in practice	X	X	X	
C	Challenges, problems and limitations with VGT in practice	X	X	X	
D	Maintenance and return on investment of VGT	X	X		
E	Long-term use of VGT in practice	X	X	X	
F	Model-based VGT combined with 2nd generation GUI-based testing	X			X
G	Failure replication	X			X

Table 1.2: *Mapping of research questions to the individual publications presented in this thesis.*

dented ability to emulate human interaction and assertions through a SUT's pictorial GUI, an ability provided by the technique's use of tools with image recognition. However, the technique's body of knowledge is limited, in particular in regards to empirical evidence for its applicability and feasibility of use in industrial practice. This lack of knowledge is the main motivator for the research presented in this thesis since such knowledge is required as decision support for industrial practitioners to evaluate if they should adopt and use the technique. Consequently, this research is motivated by an industrial need for a flexible and cost-effective GUI-based test automation technique that can emulate end user behavior with at least equal defect-finding ability as manual testing but with lower test execution time. From an academic point of view, the research is also motivated since it provides additional empirical evidence from industry regarding the adoption, use and challenges related to automated testing.

Research Objective: The objective of this thesis is to identify empirical evidence for, or against, the applicability and feasibility of VGT in industrial practice. Additionally, to identify what challenges, problems and limitations that impede the technique's short and long-term use. Hence, an overall view of the current state-of-practice of VGT, including alternative and future application areas for the technique. Consequently, knowledge that can be used for decision support by practitioners and input for future academic research.

Research questions: The research objective was broken down into four research questions presented below together with brief descriptions of how they were answered. Further, Table 1.2 presents a mapping between each research question and the papers included, and presented later, in this thesis.

RQ1: What key types of contexts and types of testing is Visual GUI Testing generally applicable for in industrial practice?

This question addresses the rudimentary capabilities of VGT, i.e. can the tech-

nique at all find failures and defects on industrial grade systems? Additionally, it aims to identify support for what types of testing VGT is used for, e.g. only regression testing of system and acceptance tests or exploratory testing as well? This question also addresses if VGT can be used in different contexts and domains, such as agile software development companies, for safety-critical software, etc. Support for this question was acquired throughout the thesis work but in particular in the studies presented in Chapters 2, 3, 4, 6 and 8, i.e. *Papers A, B, C, E and G.*

RQ2: To what extent is Visual GUI Testing feasible for long-term use in industrial practice?

Feasibility refers to the maintenance costs and return on investment (ROI) of adoption and use of the technique in practice. This makes this question key to determine the value and long-term industrial usability of VGT. Hence, if maintenance is too expensive, the time to positive ROI may outweigh the technique's benefits compared to other test techniques and render the technique undesirable or even impractical in practice. This question also concerns the execution time of VGT scripts to determine in what contexts the technique can feasibly be applied, e.g. for continuous integration? Support for this research question was, in particular, acquired in three case studies at four different companies, presented in Chapters 3, 5, and 6, i.e. *Papers B, D and E.*

RQ3: What are the challenges, problems and limitations of adopting, using and maintaining Visual GUI Testing in industrial practice?

This question addresses if there are challenges, problems and limitations (CPLs) associated with VGT, the severity of these CPLs and if any of them prohibit the technique's adoption or use in practice. Furthermore, these CPLs represent pitfalls that practitioners must avoid and therefore take into consideration to make an informed decision about the benefits and drawbacks of the technique, i.e. how the CPLs might affect the applicability and feasibility of the technique in the practitioner's context. To guide practitioners, this question also includes finding guidelines for the adoption, use and long-term use of VGT in practice.

Results to answer this question were acquired primarily from three case studies that, fully or in part, focused on CPLs associated with VGT, presented in Chapters 3, 4 and 6, i.e. *Papers B, C and E.*

RQ4: What technical, process, or other solutions exist to advance Visual GUI Testing's applicability and feasibility in industrial practice?

This question refers to technical or process oriented solutions that improve the usefulness of VGT in practice. Additionally, this question aims to identify future research directions to improve, or build upon, the work presented in this thesis.

Explicit work to answer the question was performed in an academic study, presented in Chapter 7, i.e. *Paper F*, where VGT was combined with 2nd generation technology to create a fully automated VGT tool. Additional support was acquired from an experience report presented in Chapter 8 (*Paper G*) where a novel VGT-based process was reported from industrial practice.

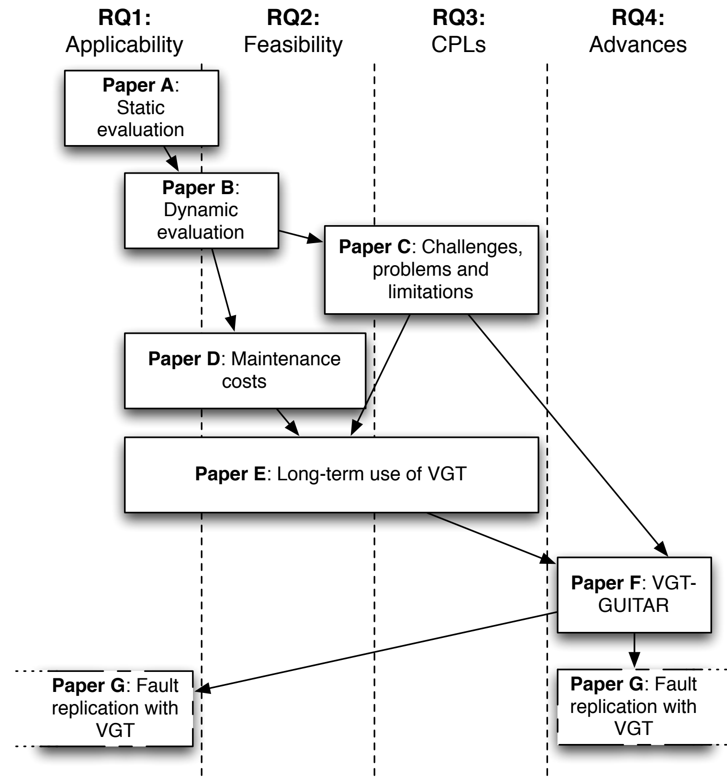

Figure 1.4: *A visualization of how the studies included in this thesis are connected to provide support for the thesis four research questions.*

1.3.2 Thesis research process

Figure 1.4 presents an overview of the incremental research process that was used during the thesis work and how included research papers are connected. These connections consist of research results or new research questions that were acquired in a study that required, or warranted, additional research in later studies.

The thesis work began with a static evaluation of VGT (*Paper A*) that provided initial support for the applicability and costs associated with VGT. Next, VGT was evaluated dynamically in an industrial project (*Paper B*) where VGT was adopted and used by practitioners. This study provided additional information about the applicability and initial results about the feasibility of VGT. In addition, challenges, problems and limitations (CPLs) were identified that warranted future research that was performed in *Paper C*. *Paper C* concluded that there are many CPLs associated with VGT but none that prohibit its industrial use. Therefore, the thesis work proceeded with an evaluation of the feasibility of VGT in an embedded study where results regarding the long-term maintenance costs and return on investment (ROI) of VGT were acquired (*Paper D*). These results were acquired through empirical work with an industrial system (Static analysis) and interviews with practitioners that had used VGT

for several months (Dynamic analysis). However, results regarding the long-term feasibility of the technique were still missing, a gap in knowledge that was filled by an interview study at a company that had used VGT for several years (*Paper E*). Consequently, these studies provided an overall view of the current state-of-practice of VGT. In addition they provided support to draw conclusions regarding the applicability (***RQ1***) and feasibility (***RQ2***) of VGT in practice but also what CPLs that are associated with the technique (***RQ3***).

Further, to advance state-of-practice, a study was performed where VGT was combined with 2nd generation technology that resulted in a building block for future research into fully automated VGT (*Paper F*)(***RQ4***). Additional support for ***RQ4*** was acquired from an experience report from industry (*Paper G*) where a novel semi-automated exploratory test process based on VGT was reported.

Combined, these studies provide results to answer the thesis four research questions and a significant contribution to the body of knowledge of VGT and automated testing.

1.3.3 Research methodology

A research methodology is a structured process that serves to acquire data to fulfill a study's research objectives [70]. On a high level of abstraction, a research process can be divided into three phases: preparation, collection and analysis (PCA). In the preparation phase the study's research objectives, research questions and hypotheses are defined, including research materials, sampling of subjects, research methods are chosen for data collection, etc. Next, data collection is performed that shall preferably be conducted with several methods and/or sources of evidence to enable triangulation of the study's results and improve the research validity, i.e. the level of trust in the research results and conclusions [70–72]. Finally, in the analysis phase, the acquired research results are scrutinized, synthesized and/or equated to draw the study's conclusions that can be both positive or negative answers to a study's research question(s).

Some research methodologies deviate from the PCA pattern and are instead said to have a flexible design. Flexible design implies that changes can be made to the design during the study to, for instance, accommodate additional, unplanned, data collection opportunities [17].

A researcher can create an ad hoc research methodology if required, but several common methodologies exist that are used in software engineering research, e.g. case studies [17], experiments [73] and action research [74].

Two research methodologies were used extensively during this thesis work: case studies and experiments. This choice was motivated by the thesis research questions and the studies' available resources. Action research was, for instance, not used because it requires a longitudinal study of incremental change to the studied phenomenon which makes it resource intensive and places a larger requirement on the collaborating company's commitment to the study. Hence, a commitment that many companies are reluctant to give to an immature research area such as VGT.

Research methodologies have different characteristics and thus, inherently, provide different levels of research validity [72]. Validity is categorized in differ-

ent ways in different research fields but in this thesis it is categorized according to the guidelines by Runeson and Höst [17], into the following categories:

- **Construct validity** - The suitability of the studied context to provide valid answers to the study's research questions,
- **Internal validity** - The strength of cohesion and consistency of collected results.
- **External validity** - The ability to generalize the study's results to other contexts and domains, and
- **Reliability/Conclusion validity** - The degree of replicability of the study's results.

Case studies provide a deeper understanding of a phenomenon in its actual context [17] and therefore have inherently high construct validity. In addition, given that a case study is performed in a well chosen context with an appropriate sample of subjects, it also provides results of high external validity. However, case studies in software engineering are often performed in industry and are therefore governed by the resources provided by the case company, which limits researcher control and can negatively affect the results internal validity.

In contrast, experiments [73] are associated with a high degree of researcher control. This control is used to manipulate the studied phenomenon and randomize the experimental sample to mitigate factors that could adversely affect the study's results. As such, experiments have inherent high internal validity but it comes at the expense of construct validity since the studied phenomenon is, by definition, no longer studied in its actual context. In addition, similar to case studies, the external validity of experimental results depend on the research sample.

Furthermore, research methodologies can be classified based on if they are qualitative or quantitative [70], where case studies are associated with qualitative data [17], e.g. data from interviews, observations, etc., and experiments are associated with quantitative data [73], e.g. measurements, calculations, etc. These associations are however only a rule of thumb since many case studies include quantitative data to support the study's conclusions [73] and experiments often support their conclusions with qualitative observations. During the thesis work, both types of data were extensively used to strengthen the papers', and the thesis, conclusions and contributions. This strength is provided by quantitative results' ability to be compared between studies, whilst qualitative data provides a deeper understanding of the results.

1.3.4 Case studies

A case study is defined as a study of a phenomenon in its contemporary context [17, 71]. The phenomenon in its context is also referred to as the study's unit of analysis, which can be a practice, a process, a tool, etc., used in an organization, company or similar context. Case studies are thereby a versatile tool in software engineering research since they can be tailored to certain

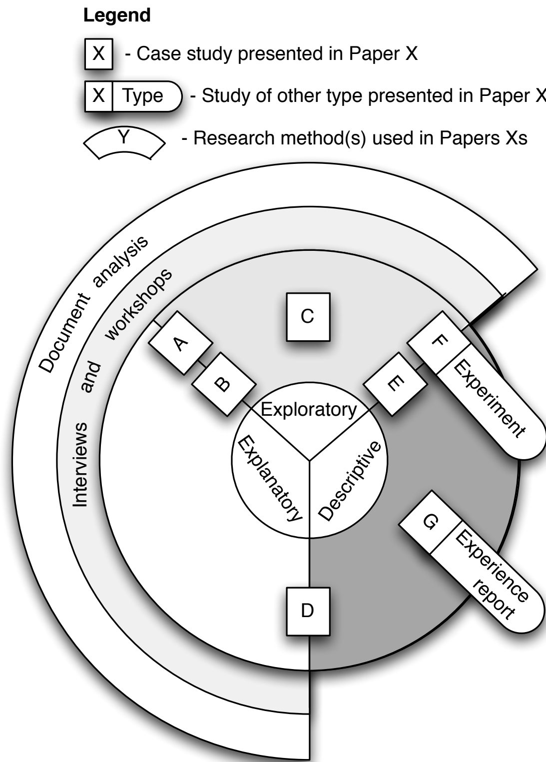

Figure 1.5: *Visualization of the categorization of each of the included papers.*

contexts or research questions and also support flexible design [17]. Additionally, case studies can be performed with many different research methods, e.g. interviews, observations, surveys, etc [17].

Further, case studies can be classified as *single* or *multiple* and *holistic* or *embedded* case studies, where single/embedded refer to the number of contexts in which the unit (holistic) or units (embedded) of analysis are studied [71].

Case study results are often anecdotal evidence, e.g. interviewees' perceptions of the research phenomenon, which makes triangulation an essential practice to ensure result validity [17, 71]. Further, case studies should be replicable, which implies that all data collection and analysis procedures must be systematic and thoroughly documented, for instance in the form of a case study protocol [71], to establish a clear chain of evidence. A more detailed discussion about analysis of qualitative data is presented in Section 1.3.6.

Case studies were the primary means of data collection for this thesis and were conducted with, or at, software development companies in Sweden. These companies include several companies in the Saab corporation, Siemens Medical and CompanyX[1]. The first case studies, reported in Papers A, B, C, were exploratory, continued with Paper D that was explanatory and concluded with Paper E that was descriptive, depicted in Figure 1.5. Hence, the thesis work

[1]For confidentiality reasons the name of this company can not be disclosed.

transitioned from exploration to explanation of the capabilities and properties of VGT to description of its use in practice. This transition was driven by the incrementally acquired results from each study, where later studies thereby aimed to verify the results of earlier studies. Figure 1.5 also includes studies that were not case studies, i.e. *Papers F and G* which were an experiment and an experience report respectively, depicted to show how they were classified in relation to the other papers included in the thesis.

Furthermore, the performed case studies were all inherently different, i.e. conducted with different companies, in different domains, with different subjects and VGT tools, which has strengthened both the construct and external validity of the thesis conclusions. Further, interviews were used for the majority of the data collection to acquire in depth knowledge about the adoption and use of VGT. However, quantitative, or quantifiable, data was also acquired since it was required to compare VGT to other test techniques in the studies, and the thesis. For instance, quantitative data was acquired to compare the performance and cost of VGT to both manual test techniques and 2nd generation GUI-based testing. However, comparisons were also made with qualitative data, such as practitioners' perceptions about benefits and drawbacks of different techniques, to get a broad view of the techniques' commonalities and differences in different contexts. Thus ensuring that the included studies' individual contributions were triangulated with data from different sources and methods to improve the results internal validity.

1.3.4.1 Interviews

Interviews are commonly used for data collection in case study research and can be divided into three different types: structured-, semi-structured and unstructured interviews [71, 75]. Each type is performed with an interview guide that contains step-by-step instructions for an interview, including the interview questions, research objectives, etc. In addition, interview guides shall include a statement regarding the purpose of the study and insurance of the interviewee's anonymity, which helps to mitigate biased or untruthful answers. Further, these types of interviews vary in strictness, which relates to the makeup of the interview guide as well as the interviewer's freedoms during an interview.

Structured interviews: Structured interviews are the most strict [71] and restrict the interviewer from asking follow up questions or ask the interviewee to clarify their answers. Therefore, considerable effort should be spent on the interview guide to test it and to ensure that the interview questions are unambiguous and valid to answer the study's research questions. Structured interview questions can be of different type but multiple-choice or forced-choice are the most common. Forced choice questions, e.g. Likert-scale questions, can be analyzed with statistics [76] but require a larger interview sample, which makes the method costly in terms of resources. Therefore, structured interviews were not used during the thesis work.

Semi-structured interviews: The second most strict type of interview is called semi-structured interviews [71], which allow the interviewer to elicit more in depth or better quality information by asking follow up questions or by clarifying questions to the interviewee. These interviews are therefore suit-

able in descriptive studies, where the studied phenomenon is partly known, or exploratory studies, where little or nothing is known about the phenomenon. In both cases, several interviews should be performed where each interview should add to the researcher's understanding of the studied phenomenon and allow the researcher to tailor each consecutive interview, i.e. change the interview questions, to acquire more in depth knowledge. However, care should be taken when changes are made to ensure that the interview results can still be triangulated, i.e. the interview questions must not diverge too much between interviews.

Semi-structured interviews were extensively used during the thesis work both to explore and describe the use of VGT and its associated CPLs [17]. Further, interview guides were always used but they were seldom changed between interviews, instead more in depth information was acquired through ad hoc follow-up questions in each interview. The baseline of common questions was kept to make triangulation of interview results easier. These interviews were all recorded, transcribed and analyzed with less rigorous qualitative analysis or Grounded Theory analysis [77], i.e. rigorous coding analysis [78].

Unstructured interviews: Finally, the least strict type of interviews are unstructured interviews [71] where interview guides are optional, but recommended, to guide the interview. This type of interview is therefore suitable for exploratory studies [17] and were primarily used in the thesis work in pre-studies to acquire contextual information about the research companies through general questions regarding the companies processes, practices, organizations, etc. In addition to context information, they also provided baseline information for additional interviews.

1.3.4.2 Workshops

Workshops are often performed with groups of participants, similar to focus groups [79] and their purpose is to explore, describe or explain a phenomenon under study [80] through discussions, brainstorming [81], activity participation, etc. As such, workshops can be a means to acquire both in depth and broad information in a short amounts of time, as discussed by Young for the purpose of requirements elicitation [82].

However, several prerequisites must be fulfilled for a workshop to be successful, for instance, the workshop participants must be a well chosen sample, i.e. with participants of varying experience, roles, age, gender, etc. This prerequisite can be challenging to fulfill in industrial studies since access to suitable participants is often restricted, for instance due to resource or scheduling constraints, etc. In addition, some constellations of participants can add bias to the results. For instance, employees can be reluctant to give their honest opinions if their managers are present in the workshop which presents a threat to the results. Additionally, if a sample is too uniform it may not provide representative results for the whole company or other domains, which is a threat to the results external validity. Another challenge with workshops is to keep them in scope of the study, especially if lively, and perhaps interesting, discussions occur, and they should therefore be moderated. Because of these challenges, workshop results must also be triangulated with, for instance, follow-up interviews, surveys, etc.

A workshop can be designed ad hoc to achieve a specific research objective, or performed with a standard workshop format, e.g. KJ-Shiba [83]. Regardless, workshops must be thoroughly planned. This planning includes sampling of suitable participants, creation of workshop materials, planning the analysis procedure, etc.

Workshops were used in several studies included in the thesis, for two reasons. First, to quickly acquire information about a research company's contexts, practices and tools. Hence, exploratory workshops with one or several participants where unstructured or semi-structured interviews and visual aids, e.g. white-board drawings, were commonly used. Second, workshops were used for result verification and triangulation where workshops began with a presentation of the study's acquired results followed by discussion and analysis of key results with the workshop's participants. These workshops, presented in *Papers B, C and E*, were well planned, with predefined workshop materials such as interview questions, presentation materials, etc., but participants were mostly sampled with convenience sampling due to resource constraints.

1.3.4.3 Other

Interviews and workshops were the primary methods used in the case studies included in this thesis. However, other methods were also used during the thesis work[2], some of which will be briefly discussed in this section.

Document analysis: Interviews and workshops acquire first degree data [71, 84], i.e. data directly from a source of information such as an interviewee. In turn, second degree data is collected indirectly from a source, for instance, through transcription of a recorded interview. However, document analysis relies on third degree data [17], which is data that has already been transcribed and potentially analyzed.

From a company perspective, document analysis can be a cost-effective method of data transference but can be a time-consuming activity for the researcher, especially in foreign or highly technical domains. Further, third degree data is created by a third person and can therefore include biases which means that document root-source analysis is required to identify who created the document, for what purpose, the age of the information, etc. [17], i.e. to evaluate the documented information's validity.

Document analysis was used in the thesis work to acquire information about the research companies' processes and practices. In particular, test specifications were analyzed to give input for empirical work with VGT at the studied companies, e.g. in *Paper A* where manual test cases at Saab AB were automated with two different VGT tools.

Further, this method can be used in a survey to conduct systematic mappings and systematic literature reviews [85] of published research papers. However, due to the limited body of knowledge on VGT, no such study was performed during the thesis work.

Surveys: Surveys are performed on samples of people, documents, software, or other group [86] for the purpose of acquiring general conclusions regarding an aspect of the sample [71]. For instance, a survey with people can

[2] In this instance, thesis work refers also to studies performed by the author but not included in the thesis.

serve to acquire their perceptions of a phenomenon, document surveys instead aim at document synthesis [85], etc.

In software engineering research, surveys are often performed with questionnaires as an alternative to structured interviews [71]. One benefit of questionnaires is that they can be distributed to a large sample at low cost but if there is no incentive for the sample to answer the questionnaire, the participant response-rate can be low, i.e. less than the rule of thumb of 60 percent that is suggested for the survey to be considered valid.

Questionnaire questions can be multiple-choice, forced-choice or open, i.e. free text. Forced choice questions are often written as Likert scale questions [76], i.e. on an approximated ratio-scale between, for instance, totally disagree and totally agree. In turn, multiple choice questions can ask participants to rank concepts on ratio-scales, e.g. with the 100 dollar bill approach [87]. However, questions can have other scales such as nominal, ordinal or interval scales [88]. These scales serve different purposes and it is therefore important to choose the right type to be able to answer the study's research questions. Regardless, questionnaire creation is a challenge since the questions must be unambiguous, complete, use context specific nomenclature, etc., to be of high quality. Therefore, like interview guides, questionnaires must be reviewed and tested prior to use.

Questionnaires were used during the thesis work to verify previously gathered results and to acquire data in association with workshops. These results were then analyzed qualitatively or with formal or descriptive statistics, discussed further in Section 1.3.6, to test the studies' hypotheses or answer the studies' research questions.

Observation: Observations are fundamental in research and can be used in different settings and performed in different ways, e.g. structured or unstructured [89]. One way to perform an observation is the fly on the wall technique, where the researcher is not allowed to influence the person, process, etc., being observed. Another type is the talk-aloud protocol, where the observed person is asked to continuously describe what (s)he is doing [17]. As such, observations are a suitable practice to acquire information about a phenomenon in its actual context and can also provide the researcher with deeper understanding of domain-specific or technical aspects of the phenomenon.

However, observation studies are associated with several threats, for instance the Hawthorne effect, which causes the observed person to change his/her behavior because they know they are being observed [90]. Therefore, the context of the observation must be taken into consideration as well as ethical considerations, e.g. how, what and why something or someone is being observed [17]. An example of unethical observation would be to observe a person without their knowledge.

Planned observation, with an observation guide, etc., was only used once during the thesis work to observe how manual testing was performed at a company. However, observations were also used to help explain results from the empirical studies with VGT, i.e. in Papers A and D.

1.3.5 Experiments

Experimentation is a research methodology [73] that focuses on answering what factor(s) (or *independent variable(s)*) that affect a measured factor of the phenomena (*the dependent variable(s)*). As such, experiments aim to compare the impact of treatments (change of the independent variable(s)) on the dependent variable(s).

Experimental design begins with formulation of a research objective that is broken down into research questions and hypotheses. A hypothesis is a statement that can be either true or false that the study will test, for instance the expected outcome of a treatment on the dependent variable. Therefore, experiments primarily aim to acquire quantitative or quantifiable data, which can be analyzed statistically to accept or reject the study's hypotheses and answer the study's research questions.

However, experiments are also affected by confounding factors [73], i.e. factors outside the researcher's control that also influence the dependent variable. These factors can be mitigated through random sampling that cancels out the confounding factors across the sample [91] such that measured changes to the dependent variable are caused only by changes to the independent variable(s).

However, in some contexts, e.g. in industry, it is not possible to randomize the studied sample and instead quasi-experiments need to be used [73, 92]. Whilst controlled experiments are associated with a high degree of internal validity but lower construct validity (due to manipulation of contextual factors), quasi-experiments have lower internal validity but higher construct validity since they are performed in a realistic context.

Further, compared to case studies, controlled experiments have high replicability, i.e. an experiment with a valid experimental procedure can be replicated to acquire the same outcome as the original experiment. It is therefore common that publications that report experimental results present the experimental procedure in detail and make the experimental materials available to other researchers.

Experiments were performed as part of two papers included in the thesis, i.e. *Papers A and F*. In *Paper A*, the applicability of VGT to test non-animated GUIs was compared to testing of animated GUIs. Additionally in *Paper F*, to compare the false test results generated by 2nd and 3rd generation GUI-based testing during system and acceptance tests.

1.3.6 Data analysis

Research methodologies and methods provide the researcher with results that sometimes are enough to support, or answer, a study's research questions. However, in most studies the results, by themselves, are not enough and they must therefore be analyzed.

Research results can be classified as qualitative, e.g. statements, and quantitative, e.g. numbers, [70], as stated in Section 1.3.3. However, regardless of data type, results must be triangulated [17, 71], which for qualitative data can be a challenge. The reason is because qualitative methods generally produce large quantities of information that are difficult to overview and synthesize. This challenge can be solved by quantifying the information through coding [78] where statements, observations, etc., are clustered in categories

defined by codes that can then be analyzed individually to identify support for the study's research question(s). This practice is key in Grounded Theory research [77] and is recommended to acquire a strong chain of evidence [71] and can be performed in different ways. However, a general recommendation is that it is performed by several people to mitigate coding bias.

During the thesis work, coding was only used for the data analysis in *Paper F* since the study relied exclusively on qualitative data. Previous studies, included in the thesis, all include empirical work and/or quantitative data, which justified the use of less stringent analysis methods since the results of these studies were triangulated with data sources of different type.

Further, quantitative data can be analyzed statistically, either with descriptive or formal methods [93,94]. Descriptive statistics are used to visualize data to provide an overview of the data, e.g. in graphs and plots, which for larger data sets can be particularly helpful since it helps establish patterns, distributions and other insights to the data. Descriptive statistics were particularly helpful in *Paper D* to visualize and draw conclusions regarding the maintenance costs and return on investment of VGT.

In contrast, formal statistics are mathematical methods that are used to compare, correlate, or evaluate data to find statistically likely patterns, e.g. to compare if two data sets are (statistically) significantly different. Formal statistical analysis was used in several of the studies included in the thesis, i.e. *Papers A, D, E and F*, most often performed with the Student T-Test, the Wilcoxon rank-sum test or the Mann-Whitney U test [95].

Further, formal statistical methods can be split into two different categories, parametric and non-parametric tests [93, 96], where parametric tests provide statistical results of high precision but are associated with more prerequisites than non-parametric tests. These prerequisites include that the data set should be normally distributed, that the sample is of suitable size, etc. However, these prerequisites are often difficult to fulfill in industrial studies and therefore it is argued that the use of parametric tests should be avoided in favor of non-parametric tests in software engineering research [96], despite the fact that non-parametric tests lower statistical precision. Due to the industrial focus of the thesis work, most formal statistical analysis performed in the included studies were non-parametric, limited primarily by lack of normal distribution in the acquired data sets.

In summary, research results can be both qualitative and quantitative and analyzed with both descriptive and formal statistics, all used during this thesis work to strengthen the validity of the studies' conclusions. Hence, quantitative data was used to statistically compare VGT tools and VGT to other test techniques, triangulated with qualitative data to explain the quantitative results and vice versa.

Paper	Name	Domain	Size (S/M/L)	City	Development process(es)	Test strategy	VGT tool
A, C and G	Saab AB	Safety-critical air-traffic management software	M	Gothenburg	Plan-driven and Agile	Manual system and acceptance testing	Sikuli (Python API)
B	Saab AB	Mission-critical military control software	M	Järfälla	Plan-driven and Agile	Manual system and acceptance testing, automated unit testing	Sikuli (Python API)
D	Saab AB	Safety-critical air-traffic management software	M	Växjö	Plan-driven and Agile	Manual system and acceptance testing	Sikuli (Python API)
D	Siemens Medical	Life-critical medical journal systems	S	Gothenburg	Agile (Scrum)	Manual scenario-based and exploratory system and acceptance testing, automated unit testing	JAutomate
E	CompanyX	Entertainment streaming application	L	Gothenburg/ Stockholm	Agile (Scrum)	Manual scenario-based and exploratory system, acceptance and user experience testing, automated unit, integration and system testing.	Sikuli (Java API)

Table 1.3: *Summary of key characteristics of the companies/divisions/groups that took part in the studies included in the thesis. In the column "size" the companies' contexts are categorized as small (S), medium (M) and large (L) where small is less than 50 developers, medium is less than 100 developers and large is more than 100 developers in total. Note that Saab AB refers to explicit divisions/companies within the Saab organization.*

1.4 Overview of publications

The following section will present summaries of the studies included in this thesis, including their research objectives, methodology, results and contributions. These studies were primarily performed in Swedish industry at companies with different organizations, processes and tools that develop both safety-critical systems as well as non-safety critical applications. An overview of these companies has been presented in Table 1.3 based on a set of characteristics that was acquired for to all companies, which includes size, domain, used test techniques, etc.

1.4.1 Paper A: Static evaluation

Paper A, presented in Chapter 2, is titled “Automated System Testing using Visual GUI Testing Tools: A Comparative Study in Industry”. The paper presents a single, embedded, case study at the safety-critical air-traffic management software development company Saab AB in the Swedish city of Gothenburg.

Research Objective: The main research objective of the study was to acquire initial support for the applicability of VGT in industrial practice. Specifically, its ability to automate system and acceptance tests for automated GUI-based regression testing.

Methodology: The case study consisted of two phases where two VGT tools, one commercial referred to as CommercialTool[3] and an open source, called Sikuli [20, 54], were evaluated. In Phase 1, static evaluation was performed of the tools to acquire their different properties. In addition, four experiments were performed to compare the tools' image recognition algorithms' ability to test animated and non-animated GUIs.

In Phase 2, 10 percent of an industrial test suite of 50 manual test cases was automated in each tool. These test cases were carefully chosen after a manual regression test of one of Saab's main product's subsystems, in continuation called the SUT. The SUT had in the order of multiple 100K lines of code, had a non-animated GUI and a comprehensive manual test suite. During test automation, measurements were taken on the development time, lines of code and script execution time. These were then compared statistically to determine if there was any statistically significant difference between the tools.

Results: Twelve (12) properties were identified in the static evaluation that showed that the two tools had different favorable properties. For instance, whilst CommercialTool had faster image recognition and support for automated test failure mitigation, Sikuli was free of charge and generally more user friendly. Further, in the experiment, CommercialTool had higher success-rate than Sikuli for non-animated GUIs but Sikuli had better success-rate for animated GUIs.

In Phase 2, the development time and execution time of the SUT's entire test suite was estimated based on the developed test scripts. These estimates showed that adoption of VGT could provide positive return on investment within one development iteration of the SUT compared to manual testing. Additionally, the estimates showed that the execution time of the suite was

[3]For reasons of confidentiality we cannot disclose the name of the tool

3.5 hours, which was an improvement of 4.5 times compared to manual testing that took 16 hours on average.

Further, none of the null hypotheses in regards to development time, lines of code and execution time could be rejected. The study therefore concludes that there is no statistical significant difference between the two tools on any of these measures. Therefore, since the tools could successfully automate the industrial test cases, the study provides initial support that VGT is applicable for automation of manual system test cases in industrial practice.

Contributions: The study's main contributions are as such:

CA1: Initial support for the applicability of VGT to automate manual scenario-based industrial test cases when performed by experts,

CA2: Initial support for the positive return on investment of the technique, and

CA3: Comparative results regarding the benefits and drawbacks of two VGT tools used in industrial practice.

This work also provided an industrial contribution to Saab with decision support regarding which VGT tool to adopt.

1.4.2 Paper B: Dynamic evaluation

Paper B, presented in Chapter 3, is titled "Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study". The paper presents a single, holistic, case study at the mission-critical military control system software development company Saab AB in the Swedish city of Järfälla.

Research Objective: The objective of the study was to acquire support for the industrial applicability of VGT when adopted and used by practitioners. Additionally, to identify challenges, solutions, costs of adoption and use of VGT and finally the practitioners' perceptions about the technique.

Methodology: The case study was performed in collaboration with Saab, where two testers used the open source VGT tool Sikuli to automate several manual test suites, at Saab referred to as acceptance test descriptions or ATDs, for the system in the continuation called the SUT. The SUT was mission critical, tested with 40 ATDs containing approximately 4000 use cases with a total execution time of 60 man weeks (2400 hours).

The study consisted of three phases, where Phase 1 was an exploratory pre-study performed to elicit the company's expectations on VGT, the company's test practices, SUT information, etc. In Phase 2, a four month project was conducted where the testers automated three out of the 40 ATDs with VGT, during which data collection was performed through telephone and e-mail due to the geographical distance between the researchers and the company. Finally in Phase 3, semi-structured interviews were used to verify the study's previous results and elicit the practitioners' perceptions about Sikuli and VGT.

Results: The study's results were divided into three parts according to the study's phases, i.e. pre-study, case-study and post-study. ***The pre-study*** provided contextual information and showed that Saab needed VGT because of the high costs associated with the SUT's manual testing.

Further, ***the case study*** showed that the VGT scripts had been implemented as 1-to-1 mappings of the manual test cases and therefore consisted of small use cases, combined in meta-models, into longer test scenarios. This architecture was perceived beneficial to lower development and maintenance costs since the modular design facilitated change and reuse of use case scripts [40].

In addition, the VGT scripts executed 16 times faster than the manual tests, identified all regression defects the manual tests found but also defects that were previously unknown to the company. These additional defects were found by changing the order of the test scripts between executions that resulted in stimulation of previously untested system states. As such, VGT improved both the test frequency and the defect-finding ability of Saab's testing compared to their manual ATD testing.

However, six challenges were found with VGT; including high SUT-Script synchronization costs, high maintenance costs of older scripts or scripts written by other testers, low (70 percent) success-rate of Sikuli's image recognition when used over a virtual network connection (VNC) (100 percent locally), unstable Sikuli behavior, etc. These challenges were solved with ad hoc solutions; for instance, minimized use of VNC, script documentation, script coding standards, etc.

Additionally, three months into the study a large change was made to the SUT that required 90 percent of the VGT scripts to be maintained. This maintenance provided initial support for the feasibility of VGT script maintenance, measured to 25.8 percent of the development cost of the suite. The scripts' development costs were also used to estimate the time to positive ROI of automating all 40 ATDs, which showed that positive ROI could be achieved within 6-13 executions of the VGT test suite.

Finally, the ***post-study*** showed that VGT was perceived as both valuable and feasible by the testers, despite the observed challenges.

Contributions: The main contributions provided by this study are as such:

CB1: Support that VGT is applicable in industrial practice when adopted and applied by practitioners in an industrial project environment,

CB2: Additional support that positive ROI can be achieved from adopting VGT in practice,

CB3: Initial support that the maintenance costs of VGT scripts can be feasible, and

CB4: Challenges and solutions related to the adoption and use of VGT in the studied project.

Consequently the study supports the contributions made by *Paper A* regarding the industrial applicability of VGT but also provides initial feasibility support for the technique. However, the study also reported several challenges with the technique that warranted further research.

1.4.3 Paper C: Challenges, problems and limitations

Paper C, presented in Chapter 4, is titled "Visual GUI Testing in Practice: Challenges, Problems and Limitations". The paper presents a multiple, holistic, case study with results from two cases. The first case was performed at the safety-critical air-traffic management software development company Saab AB in the Swedish city of Gothenburg, referred to as Case 1, whilst the second case was the case presented in Paper B, referred to as Case 2.

Research Objective: The objective of the study was to identify challenges, problems and limitations (CPLs) associated with the adoption and use of VGT in practice and general solutions to these CPLs. A secondary objective was also to find support for the ROI of adopting VGT.

Methodology: Both cases were performed independently of each other with engineers that transitioned manual test suites into VGT. This design allowed contextual CPLs and solutions to be distinguished from general CPLs and solutions. CPLs were primarily collected in Case 1, triangulated with results from Case 2, and systematically analyzed to determine the CPLs origin, generalizability, commonality to other CPLs, perceived impact, etc. These properties were then used to classify the CPLs into three tiers of varying abstraction.

In addition, metrics on development costs and execution time were acquired, which were used to estimate the time to positive ROI of adopting VGT in comparison to manual testing, in both cases.

Results: 58 CPLs were identified in the study that were classified into three tiers with 26 unique CPL groups on the lowest tier (Tier 3), grouped into eight more abstract groups (Tier 2) and finally related to three top tier CPL root causes (Tier 1). Further, 34 out of the 58 CPLs related to the tested system, 14 to the test tool (Sikuli) and 10 to third party software (VNC and simulators). Hence, more than twice of the CPLs were associated with the SUT compared to the VGT tool.

The Tier 3 CPLs were also classified into six themes to analyze their impact on VGT, which, based on the study's results and related work, showed that the CPLs had varying impact but were also common to other automated and manual test techniques. This analysis also served to identify four general solutions to mitigate half of the study's identified CPLs. These solutions included systematic development of failure and exception handling, script documentation, minimized use of remote script execution over VNC and systematic SUT-script synchronization.

In addition, measured development costs were used to calculate the time to positive ROI of adopting VGT in the two cases, which showed that positive ROI would be achieved after 14 executions of the VGT suite in Case 1 and after 13 executions in Case 2. Additionally, based on the results and related work [16], a theoretical ROI cost model was created to serve for future research into the feasibility of VGT, which would, in addition to development costs, also take maintenance costs into consideration.

Finally, the study provided additional support for the defect finding ability of VGT since nine, in total, previously unknown defects were reported, three in Case 1 and six in Case 2.

Contributions: The main contributions of this work are as such:

CC1: 29 unique groups of challenges, problems and limitations (CPLs) that affect the adoption or application of VGT in industrial practice,

CC2: Four general solutions that solve or mitigate roughly half of the identified CPLs, and

CC3: Results regarding the development costs, execution time, defect-finding ability and ROI of the use of VGT in industrial practice.

Additionally, the paper provides a general contribution to the body of knowledge of automated testing that currently only holds limited empirical support regarding CPLs [97].

1.4.4 Paper D: Maintenance and return on investment

Paper D, presented in Chapter 5, is titled "Maintenance of Automated Test Suites in Industry: An Empirical study on Visual GUI Testing". The paper presents a multiple, holistic, case study with results from the two companies Saab AB in Växsjö and Siemens Medical.

Research Objective: The objective of the study was to evaluate the maintenance costs associated with VGT scripts. Hence, results essential to support the feasibility of the use of VGT in industrial practice.

Methodology: The study was divided into two phases performed at Siemens Medical and Saab AB that were chosen through convenient sampling since companies that have used VGT for a longer period of time are rare.

Phase 1 was exploratory where three semi-structured interviews were performed to elicit practitioners' perceptions about VGT, VGT maintenance and experienced CPLs. At the time of the study, Siemens Medical had transitioned 100 out of 500 manual test cases into VGT with the VGT tool JAutomate [67] for the purpose of lowering the costs associated with manual scenario-based testing and to raise the quality of one of the company's systems.

In Phase 2, an empirical study was performed at Saab AB where a degraded [16] VGT suite was maintained by a researcher and a developer in two steps. First, the VGT suite was maintained for another version of the SUT which gave insights into the worst case maintenance costs of VGT suites. Second, the maintained VGT suite was migrated to a close variant to the SUT which gave insights into the costs of frequent maintenance of VGT suites. Fifteen (15) representative test scripts were maintained in total during the study, where representativeness was determined through analysis of the VGT suite and manual test specifications for the SUT. During the maintenance, measurements were taken on maintenance time per script, division of maintenance of images and script logic, number of found defects and script execution time. These measurements were then compared statistically to evaluate:

1. The difference in maintenance costs of frequent and infrequent maintenance.
2. The difference in maintenance costs of images and scripts, and
3. The difference between VGT development and maintenance costs,

Finally, the measurements were visualized in the theoretical cost model developed in *Paper C*.

Results: Statistical analysis of the acquired measurements provided several insights into the costs associated with VGT script development and maintenance. First, the costs of frequent maintenance was found to be statistically significantly lower than infrequent maintenance. Second, the maintenance costs of logic and images are equal in degraded VGT test suites but image maintenance is significantly lower if the test suite is maintained frequently. Finally, the maintenance cost of a script, per iteration of maintenance, is lower than the development cost of the script, regardless of frequent or infrequent maintenance.

Further, eight defects were identified during the study, some during the maintenance effort and others by running the VGT scripts, which provides further support for the defect finding ability of VGT.

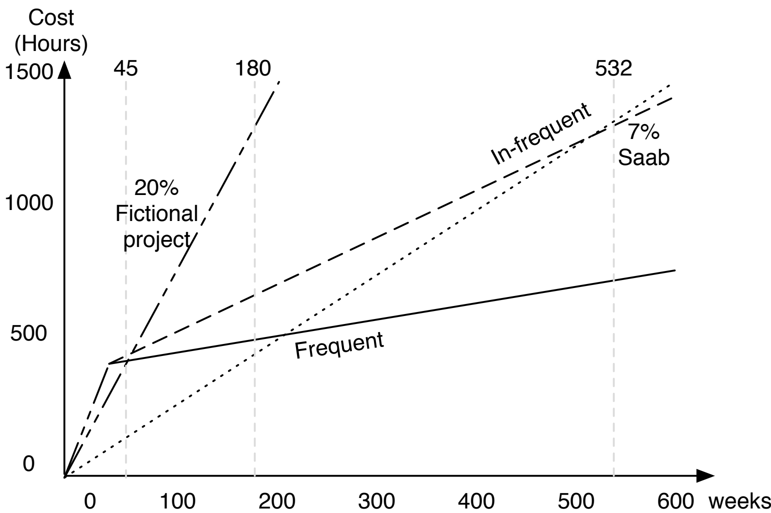

Figure 1.6: *Model of the measured development and maintenance costs of VGT compared to the costs of manual testing.*

Figure 1.6 presents a visualization of the time to positive ROI of VGT adoption and use compared to manual regression testing based on extrapolated cost data from the study. In the figure, VGT script development and frequent maintenance is shown with a ***solid line*** and VGT script development and infrequent maintenance with a ***long-dashed line***. Further, manual testing at Saab, which was seven percent of the total development costs of the project, which had 60 week iterations, is shown with a ***short-dashed line*** to be compared to a fictional project with 20 percent manual testing shown with a ***mixed long- and short-dashed line***. ***Gray, dashed, vertical lines*** show when positive ROI is reached in the different cases.

In the fictional project, positive ROI is reached in 45 weeks, i.e. within one development iteration of the project. However, in Saab AB's context, positive ROI would only be reached in 180 weeks if frequent maintenance was used and in an infeasible 532 weeks with infrequent maintenance. Hence, VGT

script maintenance is feasible but a VGT suite requires frequent maintenance if positive ROI is to be reached in a reasonable amount of time. Additionally, the time to positive ROI is dependent on the amount of manual testing performed in a project, i.e. in a project with more manual testing, positive ROI is reached faster. These results were supported by the interviews at Siemens Medical (Phase 1) where VGT script maintenance was associated with substantial cost and required effort, i.e. up to 60 percent of the time spent on VGT each week. However, despite these challenges, the technique was still considered beneficial, valuable and mostly feasibly by the practitioners.

Contributions: The main contributions of this study are as such:

CD1: That maintenance of VGT scripts is feasible in practice, shown both through qualitative and quantitative results from two companies with two different VGT tools,

CD2: That maintenance of a VGT script:

- (a) when performed frequently is significantly less costly than when performed infrequently,
- (b) is significantly less costly per iteration of maintenance than development, and
- (c) images are significantly less costly than maintenance of script logic.

CD3: That VGT scripts provide value to industrial practice, shown both with qualitative statements from practitioners and the technique's defect finding ability where eight defects were found using the technique, and

CD4: A ROI cost model based on actual data from industrial practice that verifies that this theoretical cost model is valid for automated test adoption and maintenance.

Hence, VGT can be feasible in practice but additional research was still warranted after the study to verify these results after long-term (years) use of VGT in practice.

1.4.5 Paper E: Long-term use

Paper E, presented in Chapter 6, is titled "On the Long-term Use of Visual GUI Testing in Industrial Practice: A Case Study". The paper presents a single, embedded, case study with results from the company CompanyX.

Research Objective: The objective of the study was to evaluate the long-term use of VGT in industrial practice, including short- and long-term challenges, problems and limitations (CPLs) and script maintenance costs. A secondary objective was to evaluate what alternative techniques that are used to VGT and to evaluate their benefits and drawbacks.

Methodology: The study was divided into three steps where Step 1 was a pre-study, at CompanyX, to acquire information about the company's use of VGT, willingness to participate in the study and what people to interview in the study (acquired through snowballing sampling [98]), etc.

In Step 2, four interviews were conducted with five employees at CompanyX that had detailed knowledge about how VGT and alternative automated

test techniques were used at the company. Additionally, the interviews were complemented with two workshops, one exploratory in the beginning of the study and one with one person to verify previously collected results and to identify the company's future plans for VGT.

Finally, VGT was statistically compared to an alternative test technique developed by CompanyX (the Test Interface) based on properties acquired in the interviews that were quantified based on the techniques' stated benefits and drawbacks.

Results: VGT was adopted at CompanyX after an attempt to embed interfaces for GUI testing (the Test interface) into the company's main application had failed due to lack of developer mandate and high costs. Further, because the application lacked the prerequisites of most other test automation frameworks, VGT became the only option. VGT was adopted with the tool Sikuli [54] and its success could be accounted to three factors:

1. The use of an incremental adoption process that began with a pilot project,
2. The use of best engineering practices to create scripts, and
3. The use of a dedicated adoption team.

Several benefits were observed with VGT, such as value in terms of found regression defects, robust script execution in terms of reported false test results, feasible script maintenance costs in most projects, support for testing of the release ready product, support for integration of external applications without code access into the tests, etc. Additionally, VGT integrated well with the open source model-based testing tool Graphwalker for model-based Visual GUI Testing (MBVGT). MBVGT made reuse and maintenance of scripts more cost-effective.

However, several drawbacks were also reported, such as costly maintenance of images in scripts, inability to test non-deterministic data from databases, limited applicability to run tests on mobile devices, etc. Because of these drawbacks, CompanyX abandoned VGT in several projects in favor of the originally envisioned "Test interface" solution which became realizable after the adoption of VGT due to VGT's impact on the company's testing culture. Hence, VGT had shown the benefits of automation which gave developers mandate to adopt more automation and create the Test interface. These interfaces are instrumented by Graphwalker models that use the interfaces in the source code to collect state information from the application's GUI components that is then used to assert the application's behavior. This approach is beneficial since it notifies the developer if an interface is broken when the application is compiled, which ensures that the test suites are always maintained.

Additionally, the Test interface has several benefits over VGT, such as better support for certain test objectives (e.g. tests with non-deterministic data), faster and more robust test execution, etc. However, the Test interface also has drawbacks, such as inability to verify that the pictorial GUI conforms to the application's specification, inability to perform interactions equal to a human user, required manual synchronization between application and scripts, lack of support for audio output testing, etc.

Analysis based on quantification of these properties, i.e. benefits and drawbacks, showed that there is no significant difference between VGT and the Test interface. Hence, both techniques have similar, and complementary, properties.

Finally, based on the study's results, results from *Papers B and C* and related work [99], a set of 14 guidelines were synthesized to provide practitioners with decision support and guidance to avoid pitfalls during adoption, use and long-term use of VGT in practice.

Contributions: The main contributions of this paper are that:

CE1: VGT can be used long-term in industrial practice, as shown by CompanyX's use of Sikuli for many years,

CE2: VGT has several benefits, including its flexible use that allows it to integrate external applications into the tests and test products ready for customer delivery,

CE3: VGT has many challenges, including robustness issues, possibly due to the immaturity of the technique's tools,

CE4: There are alternatives to VGT in practice with benefits such as higher robustness but with drawbacks that they do not verify that the pictorial GUI conforms to the system's specification, etc.

CE5: A set of 14 guidelines to support the adoption, use and long-term use of VGT in industrial practice.

The study thereby complements the missing results from *Paper D* regarding the long-term feasibility of VGT in practice.

1.4.6 Paper F: VGT-GUITAR

Paper F, presented in Chapter 7, is titled "Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: an Empirical Study". The paper presents a combined study with an experiment and a case study performed in an academic setting.

Research Objective: The objective of the study was to compare the differences in fault-finding ability of 2nd and 3rd generation (VGT) tools with respect to false test results for system and acceptance tests. A secondary objective was to combine the tool GUITAR with Sikuli into a hybrid tool called VGT-GUITAR and evaluate the two tools' capabilities on open source software.

VGT-GUITAR: VGT-GUITAR is an experimental tool that was developed based on the tool GUITAR's GUI ripping and MBT functionality [58], explained in Section 1.2.3. VGT-GUITAR extends GUITAR's ripper with bitmap ripping to acquire screenshots of the SUT's GUI components. These screenshots are then used during replay of test cases, generated by GUITAR, in a VGT driver (a Sikuli script) to interact with the SUT's pictorial GUI rather than by hooking into the SUT. For additional detail about the tool, see Chapter 7.

Methodology: The study began with an experiment where a custom built GUI-based application was mutated using GUI mutation operators, defined

during the study, to create 18 faulty versions of the application. A test suite was then generated for the original version of the application that was executed with GUITAR and VGT-GUITAR (Independent variable) on each mutant to measure the number of correctly identified mutants, false positives and false negative test results (Dependent variables). The dependent variables were then analyzed to compare the two techniques in terms of reported false positives and negatives for system and acceptance tests, where system tests evaluated the SUT's behavior whilst acceptance tests also took the SUT's appearance into account. In addition the execution time of the scripts in the two tools were recorded and compared.

The study was concluded with a case study where GUITAR and VGT-GUITAR were applied on three open source applications to identify support for the tools' industrial applicability.

Results: Statistical analysis of the experiment's results showed that 3rd generation scripts report statistically significantly more false positives for system tests than 2nd generation tools and that 2nd generation tools report statistically significantly more false negative results for acceptance tests. These results could be explain by observations of the scripts' behavior on different mutants and relate to how the two techniques stimulate and assert the SUT's behavior, i.e. through hooks into the SUT or by image recognition. As an example, if the GUI's appearance was changed such that a human could still interact with it, e.g. by making a button larger, the 3rd generation scripts would report a false positive result since the image recognition would fail. However, the 2nd generation scripts would pass since the hook to the button still remained. In contrast, if the GUI's appearance was changed such that a human could not interact with it, e.g. by making a button invisible, the 2nd generation scripts would produce a false negative since the hook allowed the script to interact with the invisible button. However, the 3rd generation scripts would successfully fail because the image recognition would not find a match. The results of the experiment therefore indicate that a combination of the 2nd and 3rd generation techniques could be the most beneficial because of their complementary behavior for system and acceptance tests.

The proceeding case study did however show that VGT-GUITAR is not applicable in industrial practice since the tool had zero percent success rate on any of the open source applications, caused by technical limitations in the tool, e.g. in reality it could not capture screenshots of all GUI components. Additionally, the test cases were generated for GUITAR that can, for instance, interact with a menu item without expanding the menu, i.e. functionality that is not supported by 3rd generation tools. Hence, further development is required to make VGT-GUITAR applicable in practice but the tool still shows proof-of-concept for fully automated 3rd generation GUI-based testing due to its successful use for the simpler application in the experiment.

Contributions: This study thereby provides the following main contributions:

CF1: Comparative results regarding the fault-finding ability of 2nd and 3rd generation GUI-based tools in terms of false test results for system and acceptance tests.

CF2: Initial support that a completely automated 3rd generation test tool

could be developed even though the tool developed in the study, VGT-GUITAR, still requires additional work to become applicable in practice.

As such, the study primarily provides an academic contribution regarding future research directions for VGT but also results regarding the applicability of different GUI-based test technique's use for system and acceptance testing.

1.4.7 Paper G: Failure replication

Paper G, presented in Chapter 8, is titled "Replicating Rare Software Failures with Exploratory Visual GUI Testing". The paper presents an experience report provided by Saab AB in Gothenburg about how VGT was used to replicate and resolve a defect that had existed in one of the company's systems for many years. As such, unlike the previously included papers, this paper did not have any research objective or methodology and this section therefore only present a summary of the report.

Experience report: The report presents how the company had received failure reports from their customers for several years regarding a defect in one of their systems that caused it to crash after long term use (3-7 months). These customer failure reports were however not sufficient to identify the defect and additional failure replication was therefore required. However, because the defect manifested so seldom in practice it was deemed too costly to resolve with manual practices, instead all customers were informed about the defect and were recommended to reboot the system with even frequency to mitigate failure.

In 2014 one of the company's developers found a way to remove the defect with a semi-automated test process that combined the principles of exploratory testing with VGT. In the process, a small VGT script was used to provide stimuli to the tested system's features in individual, mutually exclusive, components (methods). After each run, the script was modified by changing, or removing, methods that interacted with features that were perceived to not contribute to the manifestation of the defect, thereby incrementally ruling out which feature(s) caused the failure. Consequently an approach common to exploratory testing; simultaneous learning, test design and test execution [41,42]. The reason for the use of VGT for the process was because of the system's legacy that restricted the use of any other test automation framework.

By using the developed process, Saab AB was able to replicate the failure within 24 hours and resolve its defect within one calendar week. The defect was a small memory allocation leak, i.e. memory was not properly deallocated after a bitmap on the GUI had been rendered, but over time the leak built up to critical levels that caused the system to crash.

Post analysis of this case showed that the defect could have been found manually, at equal cost to the VGT approach but this defect analysis had to be performed by an engineer with specific technical knowledge about the system, which only a few developers at the company possessed. As such, this case shows that automated testing can provide value to a company in terms of quality gains rather than lowered costs.

Contributions: The main contributions of this experience report are as such:

CG1: A success-story from industrial practice that shows that VGT can be used to replicate and resolve non-frequent and nondeterministic defects, and

CG2: A case where automated testing was paired with manual practices to create a novel, semi-automated, test practice, implying that similar processes can be achieved with other, already available, test frameworks in practice.

Additionally this case provides implicit support of the benefits of collaboration between academia and industry since it was academic transfer of VGT knowledge to Saab AB that resulted in the company's success story.

1.5 Contributions, implications and limitations

The objective of the thesis work was to find empirical evidence for, or against, the applicability and feasibility of VGT in industrial practice. In particular, what types of testing VGT can be used for, what the maintenance costs associated with VGT scripts are and what challenges, problems and limitations (CPLs) are associated with the short and long-term use of the technique in practice. Additionally, the research aimed to find ways to advance the industrial applicability of VGT and outline areas of future VGT research.

Evidence to fulfill this objective were collected through an incremental research process that included studies in academia and Swedish industry with several VGT tools and research methods. The individual contributions of these studies can be synthesized to answer this thesis research questions, as mapped in Table 1.4, which represent four key contributions:

1. Empirical evidence for the industrial applicability of Visual GUI Testing for different types of testing, i.e. regression, system, and acceptance testing of both deterministic and non-deterministic defects, and in different contexts, i.e. for daily continuous integration, for safety-critical and non-safety critical software.

2. Empirical evidence for the industrial feasibility of Visual GUI testing, including associated script maintenance costs and reasonable time to positive return on investment, given that frequent maintenance is used and a suitable amount of effort is spent on manual testing prior to VGT adoption.

3. Empirical evidence that there are challenges, problems and limitations associated with Visual GUI Testing that affect the adoption, short and long-term use of the technique in industrial practice, and

4. Technical and process solutions to advance Visual GUI Testing's industrial applicability, currently and in the future.

Together, these four contributions lets us draw the conclusion that VGT fulfills the industrial need for a flexible GUI-based test automation technique and is mature enough for widespread use in industrial practice. This conclusion is of particular value to companies that have GUI-based systems that

lack the prerequisites, e.g. specific technical interfaces, required by other test automation frameworks since VGT finally provides these companies with the means to automate tests to lower cost and raise software quality. However, there are still many challenges, problems and limitations (CPL) associated with VGT that are pitfalls that could prohibit the successful adoption, or longer term use, of the technique in practice. Pitfalls that must be taken into consideration by adopting companies and addressed, and mitigated, by future academic research.

The continuation of this section will present the detailed syntheses of the included research papers' individual contributions and how they provide support for the thesis objective and main conclusion.

1.5.1 Applicability of Visual GUI Testing in practice

VGT is first and foremost a test technique and its applicability must therefore be judged on its ability to find defects. Ample support for the defect-finding ability of VGT was provided in the thesis from *Papers B, C, D and indicated in E*, where it was reported that VGT identified all defects found by manual scenario-based regression testing but also new defects that practitioners said would not have been found without VGT. Hence, the technique can find defects with equal, or even greater, efficiency than manual, scenario-based, system testing. As such, VGT provides concrete value in practice and a suitable complement to existing test techniques, a conclusion also supported by explicit statements from practitioners in *Papers B, D and E.* Additionally, the experience report in *Paper G* shows that VGT can be used to find unknown, non-deterministic and infrequent defects.

Further, support for the thesis conclusions were acquired with several different VGT tools, i.e. Sikuli [54], JAutomate [67] and CommercialTool. Different benefits and drawbacks were identified with the tools but their core functionality, i.e. image recognition, make them equally applicable in practice. Additionally, image recognition is what provides VGT with its main benefit, its flexibility to test almost any GUI-driven system regardless of implementation language, operating system or even platform. This provides industrial practitioners with unprecedented ability to automate not only their SUT's but also the SUT's environment, e.g. simulators, external software, etc. Thereby allowing test cases that previously had to be performed manually to be automated, a statement supported by *Papers A, B, C and E.* However, these conclusions assume that the SUT has an accessible pictorial GUI, i.e. VGT has limited or no usability for systems that lack GUI's, such as server or general backend software.

P.	**ID**	**Contribution summary**	**RQ1**	**RQ2**	**RQ3**	**RQ4**
A	**CA1**	VGT is applicable for automation of manual scenario-based industrial test cases	X			
	CA2	Initial support for the positive return on investment of VGT		X		
	CA3	Comparative results on benefits and drawbacks of two VGT tools	X		X	
B	**CB1**	VGT applicable in an industrial project environment	X			
	CB2	Positive ROI achievable after adoption of VGT in practice		X		
	CB3	Initial support that the maintenance costs of VGT scripts can be feasible		X		
	CB4	Challenges and solutions related to the adoption and use of VGT			X	
C	**CC1**	29 unique groups of challenges, problems and limitations (CPLs) that affect VGT			X	
	CC2	Four general solutions that solve or mitigate roughly half of the identified CPLs	X			
	CC3	Development costs, execution time, defect-finding ability and ROI of VGT	X	X		
D	**CD1**	Maintenance of VGT scripts is feasible in practice		X		
	CD2	That maintenance of VGT (frequent, images, maintenance) is cost-effective		X		
	CD3	VGT scripts provide value to industrial practice (e.g. finds defects)	X			
	CD4	A ROI cost model based on data from industrial practice		X		
E	**CE1**	VGT can be used long-term in industrial practice	X	X		
	CE2	VGT has several benefits in industrial practice	X			
	CE3	VGT has many challenges			X	
	CE4	There are alternatives to VGT in practice with some benefits over VGT	X			
	CE5	14 guidelines to support the adoption, use and long-term use of VGT in industrial practice	X	X	X	
F	**CF1**	Comparative results regarding 2nd and 3rd generation GUI-based tools' abilities	X			X
	CF2	Initial support for completely automated 3rd generation testing				X
G	**CG1**	VGT is able to replicate and resolve non-frequent and nondeterministic defects	X			
	CG2	VGT can be paired with manual practices to create novel, semi-automated, test practices				X
		Sum	**11**	**8**	**4**	**3**

Table 1.4: *Mapping of the individual contributions presented in Section 1.4 to the thesis research questions.* ***P*** *- Paper,* ***ID*** *- Identifier of contribution,* ***Cont.*** *- Contribution,* ***RQX*** *- Research question X,* ***CPLs*** *- Challenges, problems and limitations.*

Furthermore, in contrast to other automated test techniques, VGT enables regression testing of acceptance tests since, as discussed in Section 1.2.1, acceptance tests only differ from system tests in terms of the domain information embedded in the test scenarios, e.g. domain information that also includes the appearance of the SUT's GUI. Hence, since image recognition allows VGT to emulate a human user, it stands to reason that acceptance tests can also be automated, a conclusion also supported by *Papers B and F*. However, scenario-based scripts can only find defects in system states that are explicitly asserted, which, currently, delimits the use of VGT to automated acceptance regression testing. Acceptance testing otherwise requires cognitive reasoning and therefore a human oracle [69], which implies that it must be performed manually by an end user.

Additionally, VGT scripts execute faster than manual scenario-based test cases, reported in *Paper B* as much as 16 times faster than manual tests. However, more importantly, VGT scripts execute almost without cost and can therefore improve test frequency of GUI-based system tests, perceivably from weekly executions to daily execution or even per code commit. As a consequence, VGT can significantly improve the frequency of quality feedback to the system's developers and raise the system's quality.

However, VGT scripts are still slow in comparison to other automated test techniques, i.e. hundreds of automated unit tests can be executed at the same time as one VGT script. This conclusion presents a potential challenge for the use of VGT in practice, especially for continuous deployment where software, on commit, is automatically tested, delivered and deployed to the software's customer, discussed further in Section 1.5.2.

Further, whilst a unit test stimulates an individual software component, a single VGT script can stimulate all components of an entire sub-system, which thereby makes VGT an efficient means of achieving software component coverage. Additionally, a unit test only provides the tester with detailed knowledge of what component is broken, it does not provide information regarding what feature of the system is broken. In contrast, a VGT script can identify what feature of the system is broken but not in what component the defect resides. This observation implies that automation is required on several levels of system abstraction to provide test coverage of both software components and features of the software. In addition, it shows the value of VGT in practice since it is the only automated test technique that asserts the SUT's behavior through the same interface as the user.

However, VGT is not a replacement for existing techniques, e.g. manual regression testing, since, despite its defect-finding ability, it can only find defective SUT behavior that is explicitly asserted. In contrast, a human tester can observe faulty system behavior regardless of where or how it manifests on the GUI. However, as presented in *Papers D and E*, VGT can, and therefore only should, be used to mitigate costly, repetitive and error-prone manual testing, complemented with manual test practices, such as exploratory testing that finds new defects [42, 43].

As such, the results provided by this thesis show that VGT is applicable in industrial practice. A conclusion supported by results regarding the tools flexibility of use, improved test execution speed and defect-finding ability over manual testing. However, the technique is slower than other automated test

techniques, suffers from immature tooling and is suggested to report statistically significantly more false positives for system tests than 2nd generation GUI-based testing. VGT should therefore be complemented with other automated test techniques to provide complete test coverage of a SUT, in particular in continuous delivery contexts.

1.5.2 Feasibility of Visual GUI Testing in practice

In order for VGT to be usable in practice it is not enough that it is applicable, it also needs to be feasible. Feasibility refers to the practical and cost-effective use of a technique over a longer period of time, which implies that the development and maintenance costs of scripts need to provide positive return on investment (ROI) compared to alternative testing practices, e.g. manual testing, over time.

VGT is best compared to manual regression testing because both techniques fulfill the same test objective and use the same types of inputs and outputs for SUT stimulation and assertion. Such a comparison is also valuable since manual GUI-based testing is the only available alternative for many companies [100], e.g. due to SUT legacy or other missing prerequisites for other automated test techniques.

However, feasibility also involves test execution time since VGT is primarily an automated regression testing technique which implies that VGT scripts should be executed frequently to provide fast feedback to developers, i.e. a practice that would be prohibited by too slow execution time.

Initial data on development costs and execution time of VGT scripts were acquired in *Papers A, B and C* and were used in the respective papers to calculate time to positive ROI. However, these results were acquired for different sized VGT suites and compared to varying manual test execution costs, which make them incomparable. Therefore the results were recalculated for the development, but not maintenance, of a fictional VGT suite of 100 test cases instead of 10, 300 and 33 test cases reported in *Papers A, B and C* respectively. These fictional development costs were then compared to the average *total time spent on manual testing per iteration* in the three cases (263 manhours). Hence, in contrast to previous work where script development time was compared to the cost of running the manual test suites. This comparison thereby identifies how many times the VGT suite needs to be executed, after development, to equal the amount of manual testing that could have been performed for the same cost, i.e. the number of executions that are required for VGT adoption to provide positive ROI. Furthermore, the execution time of each fictional test suite was calculated that gives insights into the frequency with which the test suites can be executed in practice, i.e. hourly, daily or on only an even less frequent basis. The inputs and results of these calculations are presented in Table 1.5.

The table shows that the development costs of a VGT suite are considerable, i.e. in the order of hundreds of hours. However, once developed, the VGT suite provides positive ROI after 2.3 test suite executions, on average, compared to the total cost of manual testing with equivalent test cases.

Additionally, the table shows that the execution time of VGT suites can be significant and we therefore conclude that execution of a full VGT suite does

Paper	Manual test costs	Script dev. time	Dev. time of 100 scripts	Script exe. time	Exe. time of 100 scripts	Positive ROI after
A	150 mh	195 min	325 hours	3 min 27 sec	5 hours 45 min	2 exe.
B	488 mh	206 min	344 hours	18-36 sec	30-60 minutes	2 exe.
C	150 mh	387 min	645 hours	27 min	45 hours	3 exe.
Mean	**263 mh**	**262 min**	**438 hours**	**10 min 4 sec**	**17 hours 15 min**	**2.3 exe**

Table 1.5: *Table that summarizes the estimated results on development time, execution time and ROI of 100 VGT test cases from the results acquired in Papers A, B and C. Script development time is* ***compared to the average time spent on manual testing in the three projects, 263 hours****. **mh.** - Man-hours, **Dev.** - Development, **Exe.** - Execution, **ROI** - Return on investment, **Min** - Minutes, **Sec** - Seconds.*

not support faster than continuous integration on a daily basis. Therefore, test script prioritization is required to run VGT tests for regression testing and continuous integration on a hourly basis. However, in comparison to industrial state-of-practice of weekly manual regression tests, VGT still provides significantly improved test frequency [40].

Additionally, as can be seen in Table 1.5 the development costs and execution time for the VGT suite reported in *Paper C* was significantly higher than in the other two cases. The reason was because the test suite was developed to be robust, which was achieved by implementing several steps of failure mitigation code in the scripts and the test suite architecture. As such, we conclude that the architecture of VGT scripts play a role for the feasibility of VGT test development, which implies that there are VGT script best practices that should be followed, e.g. modularized test script design, scripts should be as short and linear as possible, etc. Further, more robust scripts take longer time to execute, which can stifle their use for continuous integration if the entire test suite needs to be executed often. As such there may exist a required tradeoff between robustness and execution time that needs to be taken into account during VGT script adoption and development.

However, the estimations presented in Table 1.5 do not take VGT script maintenance into account. VGT maintenance was evaluated explicitly in *Papers D and E*, where *Paper D* provided support for the feasibility of VGT script maintenance in two parts. First, the study showed, with statistical significance, that frequent maintenance of a VGT suite is less costly than infrequent maintenance. Additionally, maintenance per script per iteration of maintenance is lower than the development cost of a script, i.e. there is value in maintaining scripts rather than to rewrite them, and the cost of maintaining images is lower than script logic. Second, the quantitative results were

visualized in a ROI cost model, presented in Section 1.4.4 in Figure 1.6. These results indicate, in a best case, that the development and maintenance costs of a VGT suite provides positive ROI within on development iteration, given that at least 20 percent of the project's cost is associated with manual testing and that maintenance is performed frequently. However, if a company currently spends less time on manual testing and if scripts are maintained infrequently, the time to positive ROI could be several years, in Saab AB's case 532 weeks (or over 10 years).

Consequently, successful long-term use of VGT has several prerequisites. First, VGT needs to be integrated into the company's development and test process and the company's organization needs to be adopted to the changed process, e.g. to facilitate the need for frequent maintenance. Second, the developed VGT suite should follow engineering best practice, i.e. be based on a modularized architecture, have suitable amounts of failure mitigation, etc [40]. Further, test scripts shall be kept as short and linear as possible to mitigate script complexity, which is also mitigated by coding standards that improve script readability. Third, test automation should first, and foremost, be performed of stable test cases since this practice mitigates unnecessary maintenance costs and aligns with the techniques' primary purpose to perform regression testing. Additional factors were reported in *Paper D*, some that are common to other automated test techniques, but it is unknown how comprehensive this set of factors is and future research is therefore required to expand this set.

In summary we conclude that VGT is feasible in industrial practice with development and maintenance costs that are significant, yet manageable and provide positive return on investment. However, there are still challenges associated with the maintenance of VGT scripts that require suitable practices, organizational change as well as technical support, to be mitigated.

#	Description	Affect	Impact	Support
1	VGT scripts (Sikuli) lock up the computer during execution	Usage	Low	E
2	VGT documentation, guidelines and APIs, are lacking	Adoption	Low	B
3	Maintenance is affected by script readability, complexity, etc	Maintenance	Low	B
4	1-to-1 (manual-script) test cases are not always suitable	Maintenance	Medium	B,C
5	Manual test specifications don't always support scripting	Adoption	Medium	B,C
6	VGT tools (Sikuli and JAutomate) are immature/not robust	Usage	Medium	A,B,C,E
7	Image recognition is volatile, fails randomly	Maintenance	Medium	A,B,C
8	Script tuning (Synchronization, image similarity) is time consuming	Maintenance	Medium	B,C,E
9	SUT deficiencies (bugs, missing functionality) prohibit scripting	Adoption	High	B,C
10	Dynamic/non-deterministic output is a challenge for VGT scripts	Usage	High	A,E
11	VGT script image maintenance costs are significant	Maintenance	High	E
12	VGT scripts (Sikuli) have limited applicability for mobile testing	Usage	High	E
13	Remote script execution (VNC) negatively affects img. rec.	Adoption	High	B,C

Table 1.6: *Summary of key reported CPLs. For each CPL, its affect and impact has been ranked. Affect refers to what the CPL affects (Adoption, Usage or Maintenance). Impact on how serious (Low, Medium or High) its presence is for a company. Column "Support" indicates in which studies the CPL was reported. The table is sorted based on impact.*

Phase	#	Guideline	Description
Adoption	**1**	**Manage expectations**	It is not suitable/possible to automate anything and everything with VGT, consider what is automated and why?
	2	**Incremental adoption**	A staged adoption process that incrementally evaluates the value of VGT is suitable to minimize cost if the technique is found unsuitable.
	3	**Use a dedicated team**	A dedicated team can identify how/when/why to use VGT.
	4	**Use good engineering**	VGT costs depend on the architecture of tests/test suites and engineering best practices should therefore be used, e.g. modularization.
	5	**Consider used software**	Different software solutions, e.g. VGT tools and third party software, should be evaluated to find the best solution for the company's needs.
Use	**6**	**Change roles**	VGT can require new roles to be trained, which is associated with additional cost.
	7	**Development process**	VGT should be integrated into the development process, e.g. definition of done, and the SUT's build process, i.e. automatic execution.
	8	**Organization**	New roles require organizational changes that can disrupt development before the new ways of working settle.
	9	**Code conventions**	Code conventions help improve readability and maintainability of the scripts.
	10	**Minimize remote tests**	For distributed systems, VGT scripts should be run locally or use VGT tools with built in remote test execution support
Long-term	**11**	**Frequent maintenance**	The test process needs to prevent test cases degradation to keep VGT maintenance costs feasible long-term.
	12	**Measure**	The costs and value of VGT should be measured to identify improvement possibilities, e.g. new ways of writing scripts.
	13	**Version control scripts**	When the number of SUT variants grow, so do the test suites and they should therefore be version controlled to ensure SUT compatibility.
	14	**SUT life-cycle**	Positive return on investment of VGT adoption occurs after at least one iteration, so how long will the SUT live?

Table 1.7: *Summary of guidelines to consider during the adoption, use or long-term use of VGT in industrial practice.*

1.5.3 Challenges, problems and limitations with Visual GUI Testing in practice

Sections 1.5.1 and 1.5.2 showed that VGT is applicable and feasible in industrial practice but also mentioned challenges, problems and limitations (CPLs) with the technique. These CPLs were primarily acquired in *Papers B, C and E* and have been summarized in Table 1.6. In the table, each CPL has been classified based on what phase of VGT it affects the most, i.e. *adoption*, e.g. adoption, implementation or creation of test scripts, *usage*, e.g. running the tests or using the tests for specific test purposes or SUTs and *maintenance*, e.g. script maintenance or long-term use of VGT in a project. Further, the impact of each CPL is classified as low, medium or high, where *low* means that it is an annoyance but requires little or no mediation, *medium* means that it has a negative effect that can be removed but requires mediation and *high* means that it has a negative effect but can only be mitigated, not removed, through mediation. Mediation, in turn, implies change to a company's procedures, organization, VGT scripts, SUT, etc.

Table 1.6 shows that many VGT CPLs relate to the technique's or its tools' immaturity, e.g. lack of robustness of both image recognition and the tools themselves. As the technology matures these should be less of a problem. Further, contextual factors, such as the test environment, seems to play an important role, e.g. scripting can be prohibited by poor manual test specifications, external applications or defects in the SUT. These observations indicate an interplay between many factors and it is therefore unlikely that any one, or a simple, solution can be found to solve all the CPLs. Instead, as discussed in Section 1.5.2, VGT requires process, organizational and technical changes to be applicable and feasible. Regardless, the results of this thesis show that most CPLs can be solved or mitigated, as presented in *Papers B and C*. Consequently, no CPL was identified that prohibits the technique's use in practice but several CPLs are considered more sever, e.g. the reported need for substantial image maintenance in *Paper E*.

To provide practitioners with support to avoid these CPLs and the pitfalls with the VGT, *Paper E* presented a set of 14 guidelines for the adoption, use and long-term use of VGT in industrial practice. These guidelines are based on best practices collected from all the studies presented in this thesis and have been summarized in Table 1.7. However, this set of guidelines is not comprehensive and future research is therefore required to expand this list.

Consequently, the results of this thesis show that there are many CPLs associated with VGT that must be considered by industrial practitioners during the adoption, use or maintenance of VGT or VGT scripts. However, these CPLs also provide an academic contribution regarding potential future research areas, i.e. future research to improve the technique's applicability and feasibility in practice.

1.5.4 Solutions to advance Visual GUI Testing

The conclusion that VGT is applicable and feasible in industrial practice opens up the possibility to also focus research on the advancement of the technique's use in practice. Advances that were studied in two of the thesis included

papers, i.e. *Papers F and G*.

In *Paper F*, initial research was performed towards fully automated VGT by creating a proof-of-concept tool, VGT-GUITAR. VGT-GUITAR was shown not to be applicable in practice due to technical limitations in the tool but the study outlines a foundation for flexible, automated, GUI-based, exploratory testing, i.e. an approach that could have considerable impact in practice to lower test related costs and improve software quality. Additionally, this approach could perceivably mitigate the development and image maintenance costs of VGT through automated acquisition of images from the SUT. Hence, a technical advancement that would improve the applicability and feasibility of the current VGT technique in practice.

Further, *Paper G* reported a novel test process for semi-automated fault replication with VGT. The process combines the practices of exploratory testing with stimuli provided by a simple VGT script and advances VGT by showing its applicability for finding infrequent and non-deterministic defects. In addition, the process provides companies with a means of improving their long-term test practices since long-term tests are generally performed over weeks or months in practice but generally without SUT stimuli. VGT could provide such stimuli on the same level of abstraction as a human user and thereby improve the representativeness of the test results for actual use of the SUT in practice.

These individual contributions imply that VGT is applicable for more than regression testing in practice, an observations that roots in its ability to emulate end-use behavior. Further, these results imply that VGT can be used in contexts when the expected output can not be acquired as an oracle, instead, a more basic oracle, e.g. a crash oracle, can be used together with a human oracle to find defects, as reported in *Paper G*. *Paper F* also indicates that expected outputs can be automatically acquired through GUI ripping but future work is required to develop and evaluate the theoretical foundation presented in *Paper F*.

In summary, solutions already exist to advance the applicability of VGT, e.g. by combining VGT with human oracles in semi-automated test practices or processes. Further, advances in tooling can, through future work, enable new and more advanced types of automated GUI-based testing based on VGT.

1.5.5 Implications

This thesis presents results with implications for both industrial practice and academia, e.g. decision support for practitioners and input for future academic research.

1.5.5.1 Implications for practice

The main implication of this thesis is that VGT can be adopted and used in industrial practice, also over longer periods of time. This implies that companies with test (or software) related problems such as lacking interfaces required by other test frameworks, e.g. due to SUT legacy, or high test related costs, etc., now have a viable option for test automation. Additionally, adoption of VGT can help improve the test automation culture in a company, i.e. endorse

or mandate process, organizational or SUT changes that enable additional test automation in a company, as reported in *Paper E.*

For companies that have test automation, VGT provides a complement to their existing testing toolbox. In particular, VGT can provide high-level test automation for companies that currently test only on lower levels of system abstraction. However, VGT is not a replacement for manual GUI-based testing, instead it provides a suitable complement to mitigate the need for repetitive manual testing. This can reduce costs for the company by releasing resources, i.e. testers, which can instead focus on other types of testing, e.g. exploratory testing. Additionally, it may also raise the enjoyment of daily work for the individual, i.e. the human tester or developer. Statements supported by interview results from *Papers B, D and E.*

Further, because VGT scripts can run more frequently than manual tests, VGT can improve system quality [40]. This is an important implication of this work because it is not only of industrial benefit, but also of benefit to society, especially for safety-critical software systems, e.g. air-traffic management and medical systems, since improved quality can imply higher safety.

Another implication is that defect identification of infrequent and non-deterministic defects, e.g. defects that only manifest after longer periods of manual system interaction, are no longer out of scope due to cost. This result also implies that companies can improve their long-term test practices with continuous, user-emulated, stimuli that improve the tests' representativeness for use of the system in practice, as shown in *Paper G.*

Consequently, VGT provides several benefits to industrial practice by improving companies' test processes and thereby their software quality, improved software quality that benefits society as a whole.

1.5.5.2 Future research

This thesis provides fundamental support to the body of knowledge on VGT regarding the technique's applicability and feasibility. As such, this research presents a stepping stone for future research to advance the technique and automated testing in practice. In this section we have divided future research into five different tracks; fundamental, technical, process, psychological and related research, and discuss how each track could be pursued.

***Fundamental*:** Fundamental future research on VGT regards additional support for the conclusions of this thesis, e.g. studies in more companies and contexts. The studies included in this thesis were performed with several companies, VGT tools and domains but more work is required to strengthen the body of knowledge of VGT and to ensure the generalizability of the results. Future work can also identify more CPLs and solutions to said CPLs as well as quantitative support for the long-term applicability of VGT. Hence, research in more types of contexts, companies and for other types of systems, e.g. web-systems, but also longitudinal research that follows the entire VGT life cycle from adoption to use to long-term use of the technique.

***Technical*:** Technical future research refers to technical advancement of the technique itself, its tools and the image recognition algorithms it is built on. These improvements could help mitigate the CPLs identified with VGT, e.g. the robustness of available VGT tools, image recognition algorithms as

well as costs associated with image maintenance.

In addition, this track refers to research into novel technical solutions based on VGT such as completely automated VGT, as outlined by *Paper F*. This research could also help solve the image maintenance CPL associated with VGT, for instance through GUI ripping which would also allow VGT scripts to easily be migrated between variants of a SUT, since, as reported in *Paper E*, test script logic can be reused between variants of an application but images cannot.

***Process*:** This track regards research into processes and practices to improve the adoption, use or maintenance of VGT, the importance of which discussed in *Papers D and E*. For instance, adoption of VGT should be performed incrementally, developed test scripts should be short and linear, the test suite should have a modularized architecture, etc. *Paper E* presented a set of best practice guidelines for VGT but additional work is required to create a more comprehensive set and to evaluate the current ones validity and impact in more companies and domains. This track also includes research into how VGT should be incorporated with other test techniques and practices in a company, e.g. when and how to use VGT to lower cost and raise software quality. As discussed in Section 1.5.1, VGT can test that a SUT's features are working correctly but not where in the code a defect is, and VGT therefore needs to be complemented with testing on lower levels of system abstraction, especially for continuous delivery and deployment [45]. However, how to efficiently combine VGT with other automated test techniques is still a subject of future research.

Consequently, this track primarily focuses on research into improving best practice guidelines for how VGT should be adopted and used in industrial practice and in different contexts. In addition, this track includes development of novel practices and processes, as presented in *Paper G*, i.e. processes that make use of VGT for semi-automated testing.

***Psychology*:** The thesis explored the enjoyment of using VGT in *Papers B and D* but enjoyment is only one factor that affect the use of a tool or technique in practice, other factors such as stress, motivation, etc. are also of interest. Hence, psychological factors that could affect practitioners' perception of the technique and which could potentially be improved through improved tooling or practices. This research could also help answer why VGT has only seen moderate adoption in practice so far and why adoption of VGT seems to facilitate adoption of additional automation in a company as reported in *Paper E*. Additionally, this research could provide further insights to guide the research in the aforementioned fundamental and process oriented research tracks, e.g. what factors to focus on to improve the technique's applicability or feasibility in practice.

***Related research*:** This track relates to generalization of the results of this thesis for other automated test techniques. For instance, this thesis reports several CPLs and solutions that are general to other automated test techniques. However, future research is required to analyze if said solutions are applicable for other test techniques in practice. Further, only a few solutions were reported for the CPLs, but it is possible that solutions that exist for other techniques can be migrated to solve VGT CPLs. Hence, cross-technical migration of solutions to common CPLs.

Research question	Internal validity	External validity	Construct validity	Reliability/ conclusion validity
RQ1: Applica-bility	High	High	High	Moderate
RQ2: Feasibil-ity	High	High	High	Moderate
RQ3: CPLs	Moderate	Moderate	High	Moderate
RQ4: Future	Moderate	Moderate	Moderate	High

Table 1.8: *Summary of the threats to validity of the thesis results for each of the thesis research questions.* ***CPLs*** *- Challenges, problems and limitations.*

In addition, this thesis showed that it is possible to expand the applicability of VGT by combining it with manual practices and human oracles, a practice that is assumed to be generalizable to other automated test techniques and thereby warrants future research. Such research is of industrial interest, and importance, since it could theoretically allow companies to reuse existing tools and techniques for new purposes, thereby expanding their usefulness and improve the company's developed software.

Finally, this track includes research of how to extend or improve other test techniques with VGT capabilities, as outlined in *Paper F* where a hybrid tool for GUI-based testing was created. This research was performed with the 2nd generation GUI-based tool GUITAR but similar development could be done for other tools, e.g. the commonly used tool Selenium [55], to allow these tools to also assert the SUT's behavior through the SUT's pictorial GUI. Thus providing the tools with a wider range of applicability in practice.

1.5.6 Threats and limitations of this research

This section presents an analysis of the threats to validity of the results and conclusions presented in this thesis. Threats to validity were analyzed based on the internal, external and construct validity as well as reliability/conclusion validity of the work [71]. A summary of the evaluated validity for each research question has been presented in Table 1.8 where validity is classified either as low, moderate or high.

1.5.6.1 Internal validity

Internal validity refers to the cohesion and coherence of the results that support a research question. This was achieved in the thesis with an incremental research process where each study was based on the results, or gaps of knowledge, from previous studies. Additionally, as can be seen in Table 1.4, the papers included in this work provide multiple support for each research

question and also results that complement each other. For instance, the quantitative results regarding the feasibility of VGT (*Papers B and D* could be triangulated by statements from practitioners that had used the technique for a longer period of time (*Papers D and E)*. Similar connections were found for results that support the applicability, for instance regarding the technique's defect finding ability, both for regression testing (*Papers B, C, D and E*) and for infrequent/non-deterministic defects, (*Paper F*). Therefore the internal validity of the conclusions for research questions 1 and 2 are considered high.

However, the internal validity of identified CPLs is only considerate moderate because different, unique, CPLs were identified in different studies. This observation implies that there could be additional CPLs that can emerge in other companies and domains.

Lastly, the internal validity regarding advances of VGT are also perceived to be moderate because these results were only provided from two studies, i.e. *Paper F and G*, which had specific focuses that are perceived narrow compared to the many possible advances to VGT as outlined in Section 1.5.5.2.

1.5.6.2 External validity

External validity refers to the generalizability of the results or conclusions for other contexts and domains. The external validity of the conclusions regarding the applicability and feasibility of VGT are considered high because the results for these conclusions were acquired in different companies and with different VGT tools. Additionally, the research results come from both companies developing safety-critical software as well as agile companies developing non-safety-critical applications.

CPLs were acquired in the same contexts as the technique's applicability and feasibility but many of the reported CPLs were context dependent and it is unknown how comprehensive the set of identified CPLs are. Therefore the external validity for this research question is only considered moderate. However, the reported practitioner guidelines were triangulated with both studies on VGT in different contexts and domains as well as related research. As such the external validity of these guidelines is considered high, but more work is required to expand and evaluate this set of guidelines in the future.

Finally, for VGT advances, the external validity of the results are only considered moderate because the thesis only includes two studies, *Papers F and G*, which provide insights into explicit advances of the technique and it is therefore unknown how valuable advances in these areas would be for different industrial contexts and domains.

1.5.6.3 Construct validity

Construct validity refers to the research context's ability to provide valid results to answer the study's research questions. Most of the studies presented in this thesis were conducted in industrial practice and they are therefore perceived to have provided results of high construct validity to research questions 1, 2 and 3. However, research question 4, had less industrial support, only *Paper G*, whilst the other main contributor to the question, *Paper F*, was performed as an experiment in an academic setting with software applications,

and tools, with limited representativeness for software in industrial practice. Therefore, the construct validity for question 4 is only considered moderate.

1.5.6.4 Reliability/conclusion validity

Reliability/conclusion validity refers to the ability to replicate the study with the same results. The majority of the studies presented in this thesis were industrial case studies which implies that none of these studies can be replicated exactly. To mitigate this threat, the research methodology of each study has been presented in detail, a practice that is perceived to allow the validity of the studies to be judged without replication, triangulate the studies' results between the studies and replicate the studies in similar contexts. Further, an effort has been made in this thesis to outline the overall research process for the thesis work. However, due to the lack of replicability, of the majority of the studies, the overall reliability/conclusion validity of this research is considered moderate with the exception of the study presented in *Paper F* that presents an academic experiment that would be replicable by another researcher provided the study's research materials.

1.6 Thesis summary

The objective of this thesis was to find evidence for, or against, the industrial applicability and feasibility of Visual GUI Testing (VGT). This objective was motivated by the industrial need for a flexible technique to test software systems at high levels of system abstraction to alleviate the need for manual testing techniques that are often costly, tedious and error-prone.

The thesis work followed an incremental research methodology that began with exploratory studies to evaluate the applicability of VGT in practice. The research proceeded with studies to explain the challenges, problems and limitations (CPLs) and maintenance costs associated with the technique and was concluded with a study to verify the previously collected results and acquire evidence for the long-term use of VGT in industrial practice. Lastly, potential advances of VGT were evaluated that also outlined new areas of research and development for, or based on, VGT.

The results of these studies show that VGT is applicable and feasible in industrial practice, where applicability was supported by its:

- Faster test execution speed and improved test frequency over manual testing,
- Equal or greater defect finding ability than manual test cases,
- Ability to identify infrequent and non-deterministic defects that are too costly to find manually,
- Ability to verify the conformance of a SUT's behavior and appearance through the SUT's pictorial GUI, and
- Flexibility of use for any system with a GUI regardless of implementation language, operating system or platform.

In turn, the feasibility of the technique was supported by:

- Positive return on investment (ROI) of VGT adoption if frequent maintenance is performed,
- Maintenance costs that per iteration of maintenance per script are significantly lower than the development cost per script,
- Script execution times that allow VGT to be used for daily continuous integration, development and delivery, which also contributes to the applicability of VGT, and
- Results from industry that show its feasible use over many months or even years.

However, acquisition of these results also uncovered many challenges, problems and limitations associated with the technique, which include, but were not limited to:

- Robustness problems associated with present-day VGT tools and the image recognition algorithms they use,
- Substantial costs associated with maintenance of images,
- Required, costly to achieve, synchronization between scripts and SUT execution, and
- Environmental factors that affect the adoption, use or maintenance of VGT scripts, but also
- 14 practitioner oriented guidelines that serve to ease the adoption and use of VGT in industrial practice.

However, none of the identified CPLs was perceived, in our studies, to prohibit the use of VGT in industrial practice.

Because of the identified support for VGT, advances to the technique itself were also evaluated. First, by combining VGT with automated GUI component ripping, model-based testing and test case generation a more fully automated VGT approach was outlined. However, only initial results were acquired but enough to warrant future research which could help mitigate the costs of development and need for image maintenance reported from industrial practice. Second, an experience report from industry reported the use of VGT in a semi-automated exploratory process, which provides a broader research contribution since it shows that automated tools and techniques can be combined with manual practices to cover additional test objectives.

In summary, this thesis shows that VGT is applicable and feasible in industrial practice with several benefits over manual testing. The thesis also provides cost information and CPLs that are pitfalls that industrial practitioners must consider to make an informed decisions about VGT adoption. As such, this work provides a clear contribution for a wider industrial adoption of Visual GUI Testing. In addition, the thesis advances the knowledge on GUI-based testing and outlines several important areas of future research.

Chapter 2

Paper A: Static evaluation

Automated System Testing using Visual GUI Testing Tools: A Comparative Study in Industry

E. Börjesson, R. Feldt

***Proceedings of the 5th International Conference on Software Testing Verification and Validation (ICST'2012), Montreal, Canada, April 17-21, 2013* pp. 350-359.**

Abstract

Software companies are under continuous pressure to shorten time to market, raise quality and lower costs. More automated system testing could be instrumental in achieving these goals and in recent years testing tools have been developed to automate the interaction with software systems at the GUI level. However, there is a lack of knowledge on the usability and applicability of these tools in an industrial setting. This study evaluates two tools for automated visual GUI testing on a real-world, safety-critical software system developed by the company Saab AB. The tools are compared based on their properties as well as how they support automation of system test cases that have previously been conducted manually. The time to develop and the size of the automated test cases as well as their execution times have been evaluated. Results show that there are only minor differences between the two tools, one commercial and one open-source, but, more importantly, that visual GUI testing is an applicable technology for automated system testing with effort gains over manual system test practices. The study results also indicate that the technology has benefits over alternative GUI testing techniques and that it can be used for automated acceptance testing. However, visual GUI testing still has challenges that must be addressed, in particular the script maintenance costs and how to support robust test execution.

2.1 Introduction

Market trends with demands for faster time-to-market and higher quality software continue to pose challenges for software companies that often work with manual test practices that can not keep up with increasing market demands. Companies are also challenged by their own systems that are often Graphical User Interface (GUI) intensive and therefore complex and expensive to test [101], especially since software is prone to changing requirements, maintenance, refactoring, etc., which requires extensive regression testing. Regression testing should be conducted with configurable frequency [38], e.g. after system modification or before software release, on all levels of a system, from unit tests, on small components, to system and acceptance tests, with complex end user scenario input data [102, 103]. However, due to the market imposed time constraints many companies are compelled to focus or limit their manual regression testing with ad hoc test case selection techniques [104] that do not guarantee testing of all modified parts of a system and cause faults to slip through.

Automated testing has been proposed as one solution to the problems with manual regression testing since automated tests can run faster and more often, decreasing the need for test case selection and thereby raising quality, while reducing manual effort. However, most automated test techniques, e.g. unit testing [14, 32], Behavioral Driven Development [105], etc., approach testing on a lower system level that has spurred an ongoing discussion regarding if these techniques, with certainty, can be applied on high-level system tests, e.g. system tests [15, 16]. This uncertainty has resulted in the development of automated test techniques explicit for system and acceptance tests, e.g. Record and Replay (R&R) [6, 106, 107]. R&R is a tool-supported technique where user interaction with a System Under Test's (SUT) GUI components are captured in a script that can later be replayed automatically. User interaction is captured either on a GUI component level, e.g. via direct references to the GUI components, or on a GUI bitmap level, with coordinates to the location of the component on the SUT's GUI. The limitation with this technique is that the scripts are fragile to GUI component change [108], e.g. API, code, or GUI layout change, which in the worst case can render entire automated test suites inept [12]. Hence, the state-of-practice automated test techniques suffer from limitations and there is a need for a more robust technique for automation of system and acceptance tests.

In this paper, we investigate a novel automated testing technique, which we in the following call visual GUI testing, with characteristics that could lead to more robust system test automation [54]. Visual GUI testing is a script based testing technique that is similar to R&R but uses image recognition, instead of GUI component code or coordinates, to find and interact with GUI bitmap components, e.g. images and buttons, in the SUT's GUI. GUI bitmap interaction based on image recognition allows visual GUI testing to mimic user behavior, treat the SUT as a black box, whilst being more robust to GUI layout change. It is therefore a prime candidate for better system and acceptance test automation. However, the body of knowledge regarding visual GUI testing is small and contain no industrial experience reports or other studies to support the techniques industrial applicability. Realistic evaluation on industrial scale

testing problems are key in understanding and refining this technique. The body of knowledge neither contains studies that compare different visual GUI testing tools or the strengths and weaknesses of the technique in the industrial context.

This paper aims to fill these gaps of knowledge by presenting a comparison of two visual GUI testing tools, one commercial referred to as Commercial-Tool[1], and one open source, called Sikuli [54], in an industrial context to answer the following research questions:

RQ1: Is visual GUI testing applicable in an industrial context to automate manual high-level system regression tests?

RQ2: What are the advantages and disadvantages of visual GUI testing for system regression testing?

To answer these questions we have conducted an empirical, multi-step case study at a Swedish company developing safety-critical software systems, Saab AB. A preparation step evaluated key characteristics of the two tools and what could be the key obstacles to applying it at the company. Dynamic evaluation of the tools was then done in an experimental setup to ensure the tools could handle key aspects of the type of system testing done at the company. Finally, a representative selection of system test cases for one of the company's safety-critical subsystems was automated in parallel with both of the tools. Our results and lessons learned give important insight on the applicability of visual GUI testing.

The paper is structured as follows; section 2.2 presents related work followed by section 2.3 that describes the case study design. Section 2.4 presents results which are then discussed in section 2.5. Section 2.6 concludes the paper.

2.2 Related Work

The body of knowledge on using GUI interaction and image recognition for automation is quite large and has existed since the early 90s, e.g. Potter [18] and his tool Triggers used for GUI interactive computer macro development. Other early works includes Zettlemoyer and Amant who explored GUI automation with image recognition in their tool, VisMap. VisMap's capabilities were demonstrated through automation of a visual scripting program and the game Solitaire [19]. These early works did however not focus on automated testing but rather automation in general with the help of image recognition algorithms.

There is also a large body of knowledge on using GUI interaction for software testing, as shown by Adamoli et al. [106] who have surveyed 50 papers related to automated GUI testing for their work on GUI performance testing. Note that we differentiate between GUI interaction for automation and GUI interaction for testing since all techniques for GUI automation are not intended for testing and vice versa.

One of the most common GUI testing approaches is Record and Replay (R&R) [6, 106, 107]. R&R is based on a two step process where user mouse

[1] For reasons of confidentiality we cannot disclose the name of the tool.

and keyboard inputs are first recorded and automatically stored in a script that the tool can then replay in the second step. Different R&R tools record user input on different GUI abstraction levels, e.g. the GUI object level or the GUI bitmap level, with different advantages and disadvantages for each level. On the top GUI bitmap level a common approach is to save the coordinates of the GUI interaction in a script, with the drawback that the script becomes sensitive to reconfiguration of GUI layout but with the advantage of making the scripts robust to API and code changes. The other R&R approach is to record SUT interaction on a lower GUI object level by saving references to the GUI code components, e.g. Java Swing components, which instead make the scripts sensitive to API and code structure change [12] but more robust to GUI layout reconfiguration.

GUI testing can also be conducted on the top GUI bitmap level with techniques that use image recognition to execute test scenarios [54], in this paper referred to as visual GUI testing. Visual GUI testing is very similar to the R&R approach but with the important distinction that R&R tools do not use image recognition and are thus more hardcoded to the exact positioning of GUI elements. In current visual GUI testing tools, the common approach is that scenarios are written manually in scripts that include images for SUT interaction in contrast to the R&R approach where test scripts are commonly generated automatically with coordinates or GUI component references. In a typical visual GUI testing script input is given to the SUT through automated mouse and keyboard commands to GUI bitmap components identified through image recognition, output is then observed, once again with image recognition, and compared to expected results after which the next sequence of input is given to the SUT, etc. The advantages of visual GUI testing is that it is impervious to GUI layout reconfiguration, API and code changes, etc., but with the disadvantage that it is instead sensitive to changes to GUI bitmap objects, e.g. change of image size, shape or color.

A different approach to GUI testing is to base it on models, e.g. generate test cases from finite state machines (FSM) [109, 110]. However, the models often need to be created manually at considerable cost and the approach often face scalability problems. Automated model creation approaches have been proposed, such as GUI ripping proposed by Memon [111].

Hence, the area of GUI interaction, automation and testing, is quite broad but limited regarding empirical studies evaluating the techniques on real-world, industrial-scale software systems. Comparative research has been done on tools that use the R&R technique [106], but, to the authors' knowledge, there are no studies that compare visual GUI testing tools or evaluate if they can substitute manual regression testing in the industrial context.

Another important test aspect is acceptance testing where user and customer requirement conformity is verified with test scenarios that emulate end user interaction with the SUT. The tests are similar to system test cases, but contain more end user specific interaction information, i.e. how the system will be used in its intended domain. Acceptance test scenarios should preferably be automated and run regularly to verify system conformity to the system requirements [38] and has therefore been subject to academic research. The academic research has resulted in both tools and frameworks for acceptance test automation, including tools for GUI-interaction [31], but to the authors'

knowledge there is no research using visual GUI testing for acceptance testing.

2.3 Case Study Description

The empirical study presented in this paper was conducted in a real-world, industrial context, in one business area of the company Saab AB, in the continuation of this paper referred to as Saab. Saab develops safety critical air traffic control systems that consist of several individual subsystems of which a key one was chosen as the subject of this study. The subsystem has in the order of 100K Lines of Code (LOC), constituting roughly one third of the functionality of the system it is part of, and is tested with different system level tests, including 50 manual scenario based system test cases. At the time of the study the subsystem was in the final phase for a new customer release that was one reason why it was chosen. Other reasons for the choice included the subsystem size in LOC, the number of manual test cases, and because it had a non-animated GUI. With non-animated we mean that there are no moving graphical components, only components that, when interacted with, change face, e.g. color. Decision support information for what subsystem to include in the study was gathered through document analysis, interviews and discussions with different software development roles at Saab.

CommercialTool was selected for this study because Saab had been contacted by the tool's vendor and been provided with a trial license for the tool that made it accessible. It is a mature product for visual GUI testing having been on the market since more than 5 years. The second tool, Sikuli, was chosen since it seemed to have similar functionality as CommercialTool and, if applicable, would be easier to refine and adapt further to the company context. The company was also interested in the relative cost benefits of the tools, i.e. if the functionality or support of CommercialTool would justify its increased up-front cost.

The methodology used in the study was divided into two main phases, shown in Figure 2.1, with three steps in each phase. Phase one of the study was a pre-study with three different steps. An initial tool analysis compared the tools based on their static properties as evaluated through ad hoc script development and review of the tools' documentation. This was followed by a series of experiments with the goal of collecting quantitative metrics on the strengths and weaknesses of the tools. The experiments also served to provide information about visual GUI testing's applicability for different types of GUIs, e.g. animated with moving objects and non-animated with static buttons and images, which would provide decision support for, and possibly rule out, what type of system to study at Saab in the second phase of the study. In parallel with these experiments an analysis of the industrial context at Saab was also conducted. Phase two of the study was conducted at Saab and started with a complete manual system test of all the 50 test cases of the studied subsystem. This took 40 hours, spread over five days, during which the manual test cases were categorized based on their level of possible automation with the visual GUI testing tools. Both of the visual GUI testing tools were then used to automate five, carefully selected, representative, test case scenarios (ten percent) of the manual test suite during which metrics on

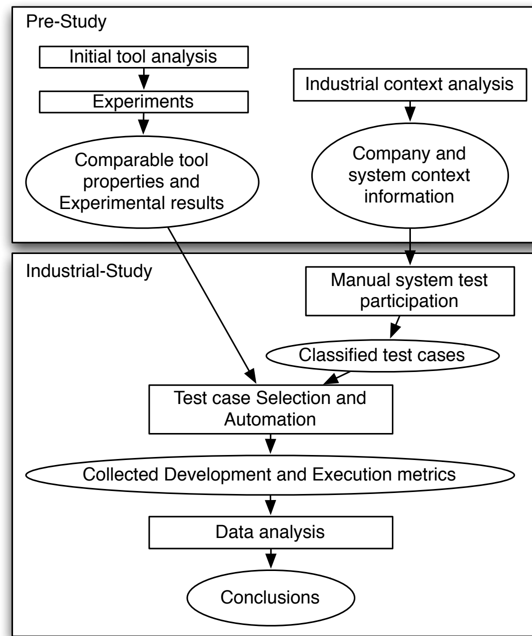

Figure 2.1: *Overview of research methodology (square nodes show activities/steps and rounded ones outcomes).*

script development time, script LOC and script execution time were collected.

In the following sections the two phases of the methodology will be described in more detail.

2.3.1 Pre-study

Knowledge about the industrial context at Saab was acquired through document analysis, interviews and discussions with different roles at the company. The company's support made it possible to identify a suitable subsystem for the study, based on subsystem size, number of manual test cases, GUI properties, criticality, etc., and to identify the manual test practices conducted at the company.

In parallel with the industrial context analysis, static properties of the studied tools were collected, through explorative literature review of the tools' documentation and ad hoc script development. The collected properties were then analyzed according to the quality criteria proposed by Illes et al. [112], derived from the ISO/IEC 9126 standard supplemented with criteria to define tool vendor qualifications. The criteria refer to tool quality and are defined as *Functionality, Reliability, Usability, Efficiency, Maintainability, Portability, General vendor qualifications, Vendor support, and Licensing and pricing.*

The tools were also analyzed in four structured experiments where scripts were written in both tools, with equivalent instructions to make the scripts comparable, and then executed against controlled GUI input. The GUI input was classified into two groups, animated GUIs and non-animated GUIs, chosen

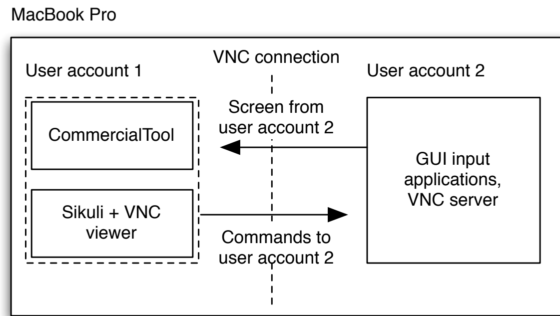

Figure 2.2: *Visualization of the experimental setup.*

to cover and evaluate how the tools perceivably performed for different types of industrial systems. The ability to handle animated GUIs is critical for visual GUI testing tools since they apply compute-intensive image recognition algorithms that might not be able to cope with highly dynamic GUIs. Eight scripts were written in total, four in each tool, and each one was executed in 30 runs for each experiment. The experiments have been summarized in the following list:

Experiment 1: Aimed to determine how well the tools could differentiate between alpha-numerical symbols by adding the numbers six and nine in a non-animated desktop calculator by locating and clicking on the calculator's buttons.

Experiment 2: Aimed to determine how the tools could handle small graphical changes on a large surface, tested by repeated search of the computer desktop for a specific icon to appear that was controlled by the researcher.

Experiment 3: Aimed to test the tools image recognition algorithms in an animated context by locating the back fender of a car driving down a street in a video clip in which the sought target image was only visible for a few video frames.

Experiment 4: Also in an animated context, aimed to identify how well the tools could track a moving object over a multi-colored surface in a video clip of an aircraft, represented by its textual call-sign, moving across a radar screen.

The four experiments cover typical functionality and behavior of most software system GUIs, e.g. interaction with static objects such as buttons or images, timed events and objects in motion, to provide a broad view of the applicability of the tools for different systems. Experiment 4 was selected since it is similar to one of the systems developed by the company.

The experiments were run on a MacBook Pro computer, with a 2.8GHz Intel Core 2 Duo processor, using virtual network computing (VNC) [113], which was a requirement for CommercialTool. CommercialTool is designed to be non-intrusive, meaning that it should not affect the performance of the

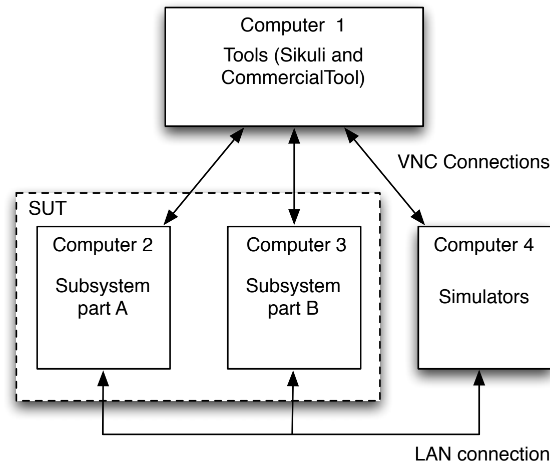

Figure 2.3: *Visualization of the test system setup.*

SUT, and to support testing of distributed software systems. This is achieved by performing all testing over VNC and support for it is built into the tool. Sikuli does not have VNC support so to equalize the experiment conditions Sikuli was paired with a third party VNC viewer application. The VNC viewer application was run on one user account connected to a VNC server on a second user account on the experiment computer, visualized in Figure 2.2.

Finally the visual GUI testing tools were also analyzed in terms of learnability since this aspect affects the technique's acceptance, e.g. if the tool has a steep learning curve it is less likely to be accepted by users [114]. The learnability was evaluated in two ad hoc experiments using Sikuli, where two individuals with novice programming knowledge, at two different occasions, had to automate a simple computer desktop task with the tool.

2.3.2 Industrial Study

The studied subsystem at Saab consisted of two computers with the Windows XP operating system, connected through a local area network (LAN). The LAN also included a third computer running simulators, used during manual testing to emulate domain hardware controlled by the subsystem's GUI. The GUI consisted primarily of custom-developed GUI components, such as buttons and other bitmap graphics, and was non-animated. During the study a fourth computer was also added to the LAN to run the visual GUI testing tools and VNC, visualized in Figure 2.3. VNC is scalable for distributed systems so the level of complexity of the industrial test system setup, Figure 2.3, was directly comparable to the complexity of the experimental setup used during the pre-study, Figure 2.2.

In the first step of the industrial study the researchers conducted a complete manual system test of the chosen subsystem with two goals. The first goal was to categorize the manual test cases as fully scriptable, partially scriptable or not scriptable based on the tool properties collected during the pre-study. The categorization provided input for the selection of representative manual test

cases to automate and showed if enough of the manual test suite could be automated for the automation to be valuable for Saab.

All the subsystem's manual test cases were scenario based, written in natural language, including pre- and post-conditions for each test case and were organized in tables with three columns. Column one described what input to manually give to the subsystem, e.g. click on button x, set property y, etc. Column two described the expected result of the input, e.g. button x changes face, property y is observed on object z, etc. The last column was a check box where the tester should report if the expected result was observed or not. The test case table rows described the test scenario steps, e.g. after giving input x, observing output y and documenting the result in the checkbox on row k the scenario proceeded on row k+1, etc., until reaching the final result checkbox on row n. Hence, the test scenarios were well defined and documented in a way suitable as input for the automation.

The second research purpose of conducting the manual system test was to acquire information of how the different parts of the subsystem worked together and what or which test cases provided test coverage for which part(s) of the subsystem. Test coverage information was vital in the manual test case selection process to ensure that the selected test cases were representative for the entire test suite so that the results could be generalized. Generalization of the results was required since it was not feasible to automate all 50 of the subsystem's manual test cases during the study.

Five test cases were selected for automation with the goal of capturing as many mutually exclusive GUI interaction types as possible, e.g. clicks, sequences of clicks, etc., to ensure that these GUI interaction types, and in turn test cases including these GUI interaction types, could be automated. GUI interaction types with properties that added complexity to the automation were especially important to cover in the five automated test cases, the most complex properties have been listed below:

1. The number of physical computers in the subsystem the test case required access to.

2. Which of the available simulators for the subsystem the test case required access to.

3. The number of run-time reconfigurations of the subsystem the test case included.

The number of physical computers would impose complexity by requiring additional VNC control code and interaction with a broader variety of GUI components, e.g. interaction with custom GUI components in subsystem part A and B and the simulators. Simulator interaction was also important to cover in the automated test cases since if some simulator interaction could not be automated neither could the manual test cases using that simulator. Run-time reconfiguration in turn added complexity by requiring the scripts to read and write to XML files. In Table 2.1 the five chosen test cases have been summarized together with which of the three properties they automate. The minimum number of physical computers required in any test case were two and maximum three whilst the maximum number of run-time configurations in any test case were also three. There were four simulators, referred to as

Test case	Physical computers	Run-time config.	Simulator
Test case 1	2	3	A
Test case 2	2	0	B
Test case 3	2	2	A
Test case 4	2	0	A
Test case 5	3	0	A

Table 2.1: *Properties of the manual test cases selected for automation. The number of physical computers does not include the computer used to run the visual GUI testing tools.*

A, B, C and D, but only simulators A and B were automated in any script because they were the most commonly used in the manual test cases and also had the most complex GUIs. In addition, simulators C and D had very similar functionality to A and B and had no unique GUI components not present in A or B and were therefore identified as less important and possible to automate.

Once the representative test cases had been selected from the manual test suite they were automated in both of the studied tools during which metrics were collected for comparison of the tools and the resulting scripts. Metrics that were collected included script development time, script LOC and script execution time.

2.4 Results

Below the results gathered during the study are presented divided into the results gathered during the pre-study and the results gathered during the industrial phase of the study.

2.4.1 Results of the Pre-study

The pre-study started with a review of the studied visual GUI testing tools' documentation from which 12 comparable static tool properties relevant for Saab were collected. The 12 properties are summarized in Table 2.2 that shows which property had impact on what tool quality criteria defined by Illes et al. [112], described in section 2.3. The table also shows what tool was the most favorable to Saab in terms of a given property, e.g. CommercialTool was more favorable in terms of real-time feedback than Sikuli. The favored tool is represented in the table with an S for Sikuli, CT for CommercialTool and (-) if the tools were equally favorable.

In the following section each of the 12 tool properties are discussed in more detail, compared between the tools and related to what tool quality criteria they impact.

Developed in: CommericalTool is developed in C#, whilst Sikuli is developed in Jython (a Python version in Java), which is relevant for the portability of the tools since CommercialTool only works on certain software platforms

whilst Sikuli is platform independent. Sikuli, being open source, also allows the user to expand the tool with new functionality, written in Jython, whilst users of CommercialTool must rely on vendor support to add tool functionality.

Script Language syntax: The script language in Sikuli is based on Python, extended with functions specific for GUI interaction, e.g. clicking on GUI objects, writing text in a GUI, waiting for GUI objects, etc. Sikuli scripts are written in the tool's Integrated Development Environment (IDE) and because of the commonality between Python and other imperative/Object-Oriented languages the tool has both high usability and learnability with perceived positive impact on script maintainability. The learnability of Sikuli is also supported by the learnability experiments conducted during the pre-study, described in Section 2.3, where novice programmers were able to develop simple Sikuli scripts after only 10 minutes of Sikuli experience and advanced scripts after an hour.

CommercialTool has a custom scripting language, modelled to resemble natural language that the user writes in the tool's IDE, which has a lot of functionality, but the tool's custom language has a higher learning curve than Sikuli script. The usability of CommercialTool is however strengthened by the script language instruction-set that is more extensive than the instruction-set in Sikuli, e.g. including functionality to analyze audio output, etc. Both Sikuli and CommercialTool do however support all the most common GUI interaction functions and programming constructs, e.g. loops, switch statements, exception handling, etc.

Supports imports: Additional functionality can be added to Sikuli by user-defined imports written in either Java or Python code to extend the tool's usability and efficiency. CommercialTool does not support user-defined imports and again users must rely on vendor support to add tool functionality.

Image representation in tool IDE: Scripts in CommercialTool refers to GUI interaction objects (such as images) through textual names whilst Sikuli's IDE shows the GUI interaction objects as images in the script itself. The image presentation in Sikuli's IDE makes Sikuli scripts very intuitive to understand, also for non-developers, which positively affects the usability, maintainability and portability of the scripts between versions of a system. In particular this makes a difference for large scripts with many images.

Real-time script execution feedback: CommercialTool provides the user with real-time feedback, e.g. what function of the script is currently being executed and success or failure of the script. Sikuli on the other hand executes the script and then presents the user with feedback, i.e. post script execution feedback. This lowers the usability and maintainability of test suites in Sikuli since it becomes harder to identify faults.

Image recognition sweeps per second: Sikuli has one image recognition algorithm that can be run five times every second whilst the image recognition algorithm in CommercialTool runs seven times every second. CommercialTool is therefore potentially more robust, e.g. to GUI timing constraints, and have higher reliability and usability, at least in theory, than Sikuli for this property.

Image recognition failure mitigation: CommercialTool has several image recognition algorithms with different search criteria that give the tool higher reliability, usability, efficiency, maintainability and portability by pro-

Property	**CommercialTool**	**Sikuli**	**Impacts**	**Favored tool**
Developed in	C#	Jython	F/P/VS	S
Script language syntax	Custom	Python	F/U/M	S
Supports imports	No	Java and Python	F/U/E/VS	S
Image representation in tool IDE	Text-Strings	Images	F/U/M/P	S
Real-time script execution feedback	Yes	No	U/M	CT
Image recognition sweeps per second	7	5	F/R/U	CT
Image recognition failure mitigation	Multiple algorithms to choose from	Image similarity configuration	F/R/U/E/M/P	CT
Test suite support	Yes	Unit tests only	F/U/M/P	-
Remote SUT connection support	Yes	No	F/U/P	-
Remote SUT connection requirement	Yes	No	F/U/P	S
Cost	10.000 Euros per license per computer	Free	U/LP	S
Backwards compatibility	Guaranteed	Uncertain	F/M/GVQ	CT

Table 2.2: *Results of the property comparison between CommercialTool and Sikuli. Column* ***Impacts****: F - Functionality, R - Reliability, U - Usability, E - Efficiency, M - Maintainability, P - Portability, GVQ - General Vendor qualifications, VS - Vendor Support, LP - Licensing and pricing. Column* ***Favored tool****: S - Sikuli, CT - CommercialTool,* ***(-)*** *- Equal between the tools*

viding automatic script failure mitigation. Script failure mitigation in Sikuli requires manual effort, e.g. by additional failure mitigation code or by setting the similarity, 1 to 100 percent, of a bitmap interaction object required for the image recognition algorithm to find a match in the GUI. Hence, Sikuli has less failure mitigation functionality that can have negative effects on usability, reliability, etc.

Test suite support: Sikuli does not have built in support to create, execute or maintain test suites with several test scripts, only single unit tests. CommercialTool has such support built in. A custom test suite solution was therefore developed during the study that uses Sikuli's import ability to run several test scripts in sequence, providing Sikuli with the same functionality, usability, perceived maintainability and portability.

Remote SUT connection support / requirement: Sikuli does not have built in VNC support, a property that is not only supported by CommercialTool but also required by the tool to operate. Sikuli was therefore paired with a third party VNC application as described in Section 2.3, to provide Sikuli with the same functionality, usability and portability as CommercialTool.

Cost: The studied tools differ in terms of cost since Sikuli is open source with no up-front cost whilst CommercialTool costs around 10.000 Euros per 'floating license' per year. A floating license means that it is not connected to any one user or computer but only one user can use the tool at a time, hence the Licensing and pricing quality criterion in this case affects the usability of CommericalTool since some companies may not afford multiple licenses while still wanting to run multiple scripts at the same time.

Backwards compatibility and support: The last property concerns the backwards compatibility of the tools, and whilst CommercialTool's vendor guarantees that the tool, which has been available in market for several years, will always be backwards compatible, Sikuli is still in beta testing and therefore subject to change. Changes to Sikuli's instruction set could affect the functionality and maintainability of the tool and scripts. This property also provides general vendor qualification information, e.g. the maturity of the vendor and the tool, which plays an important part for tool selection and tool adoption in a company, e.g. that CommercialTool may be favored because it is more mature and the tool vendor can supply support etc.

The second part of the pre-study consisted of four structured experiments, described in Section 2.3 and their results are summarized in Table 2.3. In the first experiment a script was developed in each tool for a non-animated desktop calculator application to evaluate CommercialTool's and Sikuli's image recognition algorithms' ability to identify alpha-numeric symbols. Sikuli only had a success rate of 50 percent in this experiment, over 30 runs, because the tool was not always able to distinguish between the number 6 and the letter C, used to clear the calculator, whilst CommercialTool had a success rate of 100 percent. In the second experiment the goal was to find a specific icon as it appeared on the desktop, hence identify a small bitmap change on a large surface, for which both tools had a 100 percent success rate. In the third experiment the goal was to identify the back fender of a car driving down a road in a video clip where the sought fender image was only visible for a few video frames, imposing a time constraint to the image recognition algorithms.

Experiment	Type	Desc.	CT success rate (%)	Sikuli success rate (%)
1	non-animated	Calculator	100	50
2	non-animated	Icon finder	100	100
3	animated	Car Finder	3	25
4	animated	Radar trace	0	100

Table 2.3: *Academic experiment results. CT stands for CommercialTool. Type indicates if the experiment was non-animated or not and Desc. describes the experiment.*

The car experiment resulted in Sikuli having a success rate of 25 percent and CommercialTool 3 percent. The final experiment required the tools to trace the call sign, a text string, of an aircraft moving over a multi-colored radar screen in a video-clip, where Sikuli had a 100 percent success rate whilst CommercialTool's success rate was 0 percent.

A summary of the pre-study results show that CommercialTool had higher success rate in the experiments with non-animated GUIs and had more built-in functionality required for automated testing in the industrial context, shown by the 12 analyzed properties. Sikuli on the other hand had higher success rate in the experiments with animated GUIs and showed to be easier to adapt, only requiring small efforts to be extended with additional functionality. In addition, Sikuli was considered marginally favored according to the tool quality criteria defined by Illes et al. and is therefore perceived as a better candidate for future research.

2.4.2 Results of the industrial study

The industrial part of the study started with the researchers conducting a complete manual system test of the studied subsystem. During the manual system test all the test cases were analyzed, as described in Section 2.3, and classified into categories. The category analysis showed that Sikuli could fully script 81%, partially script 17% and not script 2% of the manual test cases. CommercialTool on the other hand could fully script 95%, partially script 3% and not script 2% of the manual test cases. The higher percentage of scripts that could be fully automated in CommercialTool was given by the tool's ability to analyze audio output, required in seven of the manual test cases. The 2% of the manual test cases that could not be scripted, in either tool, were hardware related and required physical interaction with the SUT.

Based on the categorization and the selection criteria, discussed in Section 2.3, five manual test cases were chosen for automation. The automation was done pair-wise in each tool, e.g. test case x was automated in one tool and then in the other tool, with the order of the first tool chosen at random for each test case. Random tool selection was used to ensure that the script development time for the script developed in the secondly used tool would not continuously

	CT			**Sikuli**			
Test case	**Dev-time (min)**	**Exe-time (sec)**	**LOC**	**Dev-time (min)**	**Exe-time (sec)**	**LOC**	**TC Steps**
ATC-1	255	111	103	105	90	212	5
ATC-2	195	405	233	200	390	228	4
ATC-3	285	390	368	260	338	345	16
ATC-4	205	80	80	180	110	92	9
ATC-5	120	90	115	150	154	169	8
Total:	17 hours 40 minutes	17.93 minutes	899 LOC	15 hours 55 minutes	18.00 minutes	1046 LOC	

Table 2.4: *Metrics collected during test case automation. CT stands for CommercialTool, ATC for automated test case and TC steps for the number of test steps in the scenario of the manual test case.*

be skewed, lowered, because challenges with the script, e.g. required failure mitigation, etc., had already been resolved when the script was developed in the first tool.

The main contributor to script development time was in the study observed to be the amount of code required to mitigate failure due to unexpected system behavior, e.g. GUI components not rendering properly, GUI components appearing on top of each other, etc. Failure mitigation was achieved through ad hoc addition of wait functions, conditional branches and other exception handling, e.g. try-catch blocks, which for each added function required extra image recognition sweeps of the GUI that also increased the script execution time. Scripts that required failure mitigation also took longer to develop since they had to be rerun more times during development to ensure script robustness. The development time required to make a script robust also proved to be very difficult to estimate because unexpected system behavior was almost never related to the test scenarios but rather a product of the subsystem's implementation. Each script was developed individually and consisted of three parts. First a setup part to cover the preconditions of the test case. The second part was the test scenario and the third part was a test teardown to put the subsystem back in a known state to prepare it for the following test case. After the five test scripts had been developed in each tool the LOC and execution time for each script was recorded, shown in Table 2.4 together with the script development time and number of steps in the corresponding manual test case scenario.

Table 2.4 shows that the total development time, LOC and execution time were similar for the scripts in both tools.

The five chosen test cases were carefully selected to be representative for the entire manual test suite for the subsystem, as described in section 2.3, to allow the collected data to be used for estimation. Estimation based on the

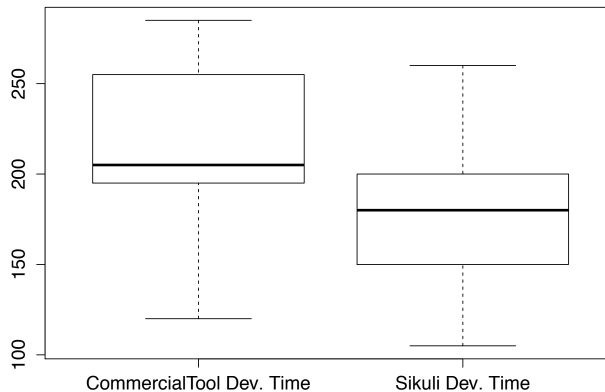

Figure 2.4: *Boxplot showing development time of the five scripts in each tool.*

average execution times, from Table 2.4, shows that the fully automated test suite for the subsystem, all 50 test cases, would run in approximately three and a half hours in each tool. A three and a half hour execution time constitutes a gain of 78 percent compared to the execution time of the current manual test suite, 16 hours, if conducted by an experienced tester. Hence, automation would constitute not only an advantage in that it can be run automatically without human input but a considerable gain in total execution time which allows for more frequent testing. Potentially tests can run every night and over weekends and shorten feedback cycles in development. In Figure 2.4 the script development time for the scripts, taken from Table 2.4, have been visualized in a box-plot that shows the time dispersion, mean development time, etc. Using the mean development time, the development time for the entire automated test suite, all 50 test cases, can be estimated to approximately 21 business days for CommercialTool and 18 business days for Sikuli. The estimated development time for the automated test suite is in the same order of time that Saab spends on testing during one development cycle of the subsystem. Hence, the investment of automating the test suite is perceived to be cost beneficial after one development cycle of the subsystem.

The data in Table 2.4 was also subject to statistical tests to see if there was any statistical significant difference between the two tools. The data was first analyzed with a Shapiro-Wilks test of the difference between the paired variables in Table 2.4, which showed that the data was normally distributed. Normal distribution allowed the data to be analyzed further with the Student t-test that had the p-value results 0.3472 for development time, 0.956 for execution time and 0.2815 for LOC. The Student t-test results were then verified with a non-parametric paired Wilcoxon test that had results with the same statistical implications. Hence, both the Student t- and Wilcoxon-tests showed that we cannot reject the null hypothesis, H_0, on a 0.05 confidence level. Therefore, it can be concluded that there is no statistical significant dif-

ference between the scripts of the studied tools in terms of development time, execution time or LOC. The statistical results are however limited by the few data points the tests were conducted on.

2.5 Discussion

Our study shows several differences between the two studied tools but that both tools were able to successfully automate 10 percent of an industrial manual system test suite, for which 98 percent of the test cases can be fully or partially automated with visual GUI testing. The open-source tool, Sikuli, had a higher percentage of test cases that could only be partially scripted since it has no current support for detecting audio output. However, this is not a major obstacle since either the audio output can be visualized, and thus tested visually, or Sikuli can be extended with Operating System (OS) system calls.

CommercialTool and Sikuli differ in terms of cost, vendor support, test functionality, script languages, etc., with impacts on different tool quality criteria, shown in Table 2.2, and are all important properties to consider for the industrial applicability of visual GUI testing. However, to show that visual GUI testing has any applicability at all in industry the most important aspect concerns the functionality of the image recognition algorithms.

The image recognition algorithms are what sets visual GUI testing apart from other GUI testing techniques, e.g. R&R, and also determine for what types of systems it is possible to apply the technique. R&R that interacts through GUI components was determined as unsuitable for the automation of the subsystem test cases since they had to interact with components not developed by Saab, e.g. interaction with custom and OS GUI components. These interactions required access to GUI component references that could not be acquired. The GUI components in the SUT, e.g. the simulators, windows in the OS, etc., did not always appear in the same place on the screen when launched. This behavior also ruled out R&R with coordinate interaction as an alternative for the study. Evaluation of visual GUI testing showed that it does not suffer from R&R's limitations and therefore works in contexts where R&R cannot be applied. Visual GUI testing is applicable on different types of GUIs, evaluated in the pre-study experiments and in industry, which showed that both studied tools had high success-rates with non-animated GUIs and that Sikuli also had good success-rate on animated GUIs as well. Hence, this study shows that visual GUI testing works for tests on non-animated GUIs and perceivably also for animated GUIs. Non-animated GUI applicability is however a subject for future deeper research.

The purpose of automation of manual tests is to make the regression testing more cost-efficient by increasing the execution speed and frequency and lower the required manual effort of executing the tests cases. Estimations based on the collected data show that a complete automatic test suite for the studied subsystem would execute in three and a half hours, which constitutes a 78 percent reduction compared to manual test execution with an experienced tester. Hence, the automated test suite could be run daily, eliminating the need for partial manual system tests, reduce cost, increase test frequency and lower

the risk of slip through of faults. Mitigation of slip through of faults is however limited with this technique by the test scenarios since faulty functionality not covered by the test scripts would be overlooked, whilst a human tester could still detect them through visual inspection. Hence, the automated scripts cannot replace human testers and should rather be a complement to other test practices, such as manual free-testing. The benefit of visual GUI testing scripts compared to a human tester in terms of test execution is that the scripts are guaranteed to run according to the same sequence every time, whilst human testers are prone to take detours and make mistakes during testing, e.g. click on the wrong GUI object, etc., which can cause faults to slip through.

Scenario based system tests are very similar to acceptance tests and based on the results of this study it should therefore be concluded as plausible to automate acceptance tests with visual GUI testing. This conclusion is supported by the research of similar GUI testing techniques, e.g. R&R, which has been shown to work for acceptance test automation [31, 107]. Further support is provided by the fact that some of the manual test cases, categorized as fully scriptable, for the studied subsystem had been developed with customer specific data. The results of this study therefore provide initial support that visual GUI testing can be used for automated acceptance testing in industry.

During the study it was established that the primary cost of writing visual GUI testing scripts was related to the effort required to make the scripts robust to unexpected system behavior. Unexpected system behavior can be caused by faults in the system, related or unrelated to the script, and must be handled to avoid that these faults are overlooked or break the test execution. Other unexpected behavior can be caused by events triggered by the system's environment, e.g. warning messages displayed by the OS. Hence, events that may appear anywhere on the screen. These events can be handled with visual GUI testing but are a challenge for R&R since the events location, the coordinates, are usually nondeterministic. Script robustness in visual GUI testing can be achieved through ad hoc failure mitigation but is a time-consuming practice. A new approach, e.g. a framework or guidelines, is therefore required to make robust visual GUI test script development more efficient. Hence, another subject for future research.

The cost of automating the manual test suite for the studied subsystem was estimated to 20 business days, which is a considerable investment, and to ensure that it is cost-beneficial the maintenance costs of the suite therefore have to be small. Small is in this context measured compared to the cost of manual regression testing, hence the initial investment and the maintenance costs have to break even with the cost of the manual testing within a reasonable amount of time. The maintenance costs of visual GUI testing scripts when the system changes are however unknown and future research is needed.

Our results show that visual GUI testing is applicable for system regression testing of the type of industrial safety critical GUI based systems in use at Saab. The technique is however limited to find faults defined in the scripted scenarios. Hence, visual GUI testing cannot replace manual testing but minimize it for customer delivery. Visual GUI testing also allows tests to be run more often and are more flexible than other GUI testing techniques, e.g. coordinate based R&R, because of image recognition that can find a GUI component regardless of its position in the GUI. Furthermore, R&R tools that require

access to the GUI components, in contrast to visual GUI testing, are not easily applicable at this company since their systems have custom-developed GUIs as required in their domain. We have also seen that visual GUI testing can be applied for automated acceptance testing. Being able to continuously test the system with user-supplied test data could have very positive results on quality.

Evaluating a technique's applicability in a real-world context is a complex task. We have opted on a multi-step case study that covers multiple different criteria that gives the company better decision support on which to proceed. Even though the test automation comparison is based on a limited number of test cases the research was designed so that these test cases are representative of the rest of the manual test suite. Still, this is a threat to the validity of our results. Our industrial partner is more concerned with the amount of maintenance that will be needed as the system evolves. If these costs are high they will seriously limit the long-term applicability of visual GUI testing.

2.6 Conclusion

In this paper we have shown that visual GUI testing tools are applicable to automate system and acceptance tests for industrial systems with non-animated GUIs with both cost and potentially quality gains over state-of-practice manual testing. Experiments also showed that the open source tool that was evaluated can successfully interact with dynamically changing, animated GUIs that would broaden the number and type of systems it can be successfully applied to.

We present a comparative study of two visual GUI testing script tools, one commercial and one open source, at the company Saab AB. The study was conducted in multiple steps involving both static and dynamic evaluation of the tools. One of the company's safety critical subsystems, distributed over two physical computers, with a non-animated GUI, was chosen and 10 percent, 5 out of 50, representative, manual, scenario-based, test cases were automated in both tools. A pre-study helped select the relevant test cases to automate as well as evaluate the strengths and weaknesses of the two tools on key criteria relevant for the company.

Analysis of the tools properties show differences in the tools functionality but overall results show that both studied tools work equally well in the industrial context with no statistically significant differences in either development time, run time or LOC of the test scripts. Analysis of the subsystem test suite show that up to 98 percent of the test cases can be fully or partially automated using visual GUI testing with gains to both cost and quality of the testing. Execution times of the automated test cases are 78% lower than running the same test cases manually and the execution requires no manual input.

Our analysis shows that visual GUI testing can overcome the obstacles of other GUI testing techniques, e.g. Record and Replay (R&R). R&R either requires access to the code in order to interact with the System Under Test (SUT) or is tied to specific physical placement of GUI components on the display. Visual GUI testing is more flexible, interacting with GUI bitmap components through image recognition, and robust to changes and unexpected behavior during testing of the SUT. Both of these advantages were important

in the investigated subsystem since it had custom GUI components and GUI components that changed position between test executions. However, more work is needed to extend the tools with ways to specify and handle unexpected system events in a robust manner; the potential for this in the technique is not currently well supported in the available tools. For testing of safety-critical software systems there is also a concern that the automated tools are not able to find defects that are outside the scope of the test scenarios, such as safety defects. Thus any automated system testing will still have to be combined with manual system testing before delivery but the main concern for future research is the maintenance costs of the scripts as a system evolves.

Chapter 3

Paper B: Dynamic evaluation

Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study

E. Alégroth, R. Feldt, H. H. Olsson

Accepted at the 6th International Conference on Software Testing Verification and Validation (ICST'2013), Luxenbourg, March 18-22, 2013.

Abstract

Visual GUI testing (VGT) is an emerging technique that provides software companies with the capability to automate previously time-consuming, tedious, and fault prone manual system and acceptance tests. Previous work on VGT has shown that the technique is industrially applicable, but has not addressed the real-world applicability of the technique when used by practitioners on industrial grade systems. This paper presents a case study performed during an industrial project with the goal to transition from manual to automated system testing using VGT. Results of the study show that the VGT transition was successful and that VGT could be applied in the industrial context when performed by practitioners but that there were several problems that first had to be solved, e.g. testing of a distributed system, tool volatility. These problems and solutions have been presented together with qualitative, and quantitative, data about the benefits of the technique compared to manual testing, e.g. greatly improved execution speed, feasible transition and maintenance costs, improved bug finding ability. The study thereby provides valuable, and previously missing, contributions about VGT to both practitioners and researchers.

3.1 Introduction

To date, there are no industrial case studies, from the trenches, that visual GUI testing (VGT) works in industry when used by practitioners, nor data to support the long-term viability of the technique. In our previous work, we have shown that VGT is applicable in industry, even for testing of safety-critical software [65]. However, previous work has been essentially driven by researchers, e.g. they applied VGT techniques, compared the resulting test cases to earlier manual efforts, and then collected feedback and refinements from the industrial practitioners. There is a risk that this type of research does not consider all the complexities and problems seen by practitioners when actually applying a technique in practice. Furthermore, researcher driven studies are often smaller in scale and cannot evaluate longer term effects such as maintenance and refactoring of the test scripts or effects on, and of, changes to the system under test (SUT). Hence, there is still a gap in VGT's body of knowledge regarding if the technique is applicable when performed by industrial practitioners in a real world development context.

In this paper we aim to bridge this gap by presenting an industrial case study from a successful project, driven entirely by industrial practitioners, with the goal to transition into VGT at the company Saab AB, subdivision security and defense solutions (SDS). The company chose VGT because of its ability to automate high system-level test cases, which previous automation techniques, e.g. unit testing [14, 32] and record and replay (R&R) [6, 106, 107], have had shortcomings in their ability to achieve. High system-level tests developed with automated unit tests have become both costly and complex, thereby spurring a discussion if the technique is applicable for anything but the low system-level testing, for which it was developed [15]. Furthermore, R&R techniques, which were developed for automation of system-level tests, are instead limited by being fragile to GUI layout and API change. Limitations that in the worst case have caused entire automated test suites to become inept [12]. Hence, the previous techniques have shortcomings in terms of flexibility, simplicity and robustness to make them long-term viable.

However, in this case study we show that VGT can overcome these limitations. Hence, showing that VGT has the capability to automate and perform industrial grade test cases that previously had to be performed manually, with equal or even greater fault finding ability, at lower cost. Capability provided by the technique's use of image recognition that, in combination with scenario based scripts, allow VGT tools to interact with any graphical object shown on the computer monitor, i.e. allowing VGT scripts to emulate a human user. In addition, the study presents the practitioners' views on using the technique, e.g. benefits, problems and limitations, when performed with the open source tool Sikuli [54]. Consequently, this work shows that VGT works for testing of real-world systems when performed by practitioners facing real-world challenges such as refactoring and maintenance of the SUT. The specific contributions of this work therefore include,

C1: An account on how the transition to VGT was successfully conducted by industrial practitioners for a real-world system.

C2: The industrial practitioners experiences and perception on the use of

VGT.

C3: Qualitative and quantitative data on costs, challenges, limitations and solutions that were identified during the VGT transition project.

Together these contributions can help both industrial practitioners in practice and guide researchers in further advancing the state-of-practice and state-of-the-art in VGT.

The continuation of this paper is structured as follows. In Section 3.2 related work is presented, followed by Section 3.3 that presents the case study methodology and data collection. Section 3.4 then presents the results and analysis of the study that are then discussed in Section 3.5. Finally the paper is concluded in Section 3.6.

3.2 Related Work

The concepts of using image recognition for GUI interaction is quite old and has been evaluated in a considerable body of knowledge. Work on using image recognition for GUI automation can be traced back to the early 90s, e.g. Potter [18] and his computer macro development tool, Triggers. Other early work in this area include the work of Zettlemoyer and Amant that used image recognition in their tool VisMap, which was used to automate the interaction with a visual scripting program as well as the game Solitaire [19]. However, this work focused on using image recognition for automation which we differentiate from testing since not all tools developed for GUI automation are intended for testing and vice versa.

The body of knowledge on using GUI interaction for testing is also considerable, e.g. shown by Adamoli et al. [106] in their paper on automated performance testing that covers 50 papers on automated GUI testing. Automated GUI testing can be performed with different techniques but the most common approach is referred to as record and replay (R&R) [6,106,107]. R&R consists of two steps. First a recording step where user input, e.g. mouse and keyboard interaction, to the system under test (SUT) is recorded in a script. In the second step, the recorded script can automatically be replayed for regression testing purposes. Different R&R tools record SUT interaction on different levels of GUI abstraction where the most common are on GUI bitmap level, i.e. using coordinates, or GUI widget level, i.e. using software references to buttons, textfields, etc. However, both approaches suffer from limitations that affect their robustness. Coordinate based R&R has the limitation that it is sensitive to GUI layout change whilst being robust to SUT code change. Widget based R&R, in contrast, is sensitive to SUT API or code structure change [12], but is instead robust to GUI layout change.

Image recognition based GUI testing with scenario based scripts, which we refer to as visual GUI testing (VGT), does not suffer from these limitations but it is only recently that the technique started to emerge in industry. One plausible explanation to this phenomenon is that the image recognition is performance intensive and it is not until now that the hardware has become powerful enough to cope with the performance requirements. VGT is a tool-supported technique, e.g. by Sikuli [54], EggPlant, etc., which conducts testing

through the top GUI bitmap level of a SUT, i.e. the actual bitmap graphics shown to the human user on a computer monitor. Hence, scenario based VGT scripts can emulate a human user and can therefore also test all applications, regardless of implementation or platform, e.g. web, desktop, mobile. In most VGT tools the scenarios have to be developed manually, but there are also tools, e.g. JAutomate, which has record and replay functionality. Typical VGT scripts are executed by first providing the SUT with input, i.e. clicks or keyboard input, after which the new state of the system is observed, using image recognition, and compared to some expected output, followed by a new sequence of inputs, etc. In contrast to previous GUI testing techniques, VGT is impervious to GUI layout change, API or even code changes. However, VGT is instead sensitive to GUI graphics changes, e.g. changes in graphics size, shape or color.

Another approach to GUI testing is to use models, e.g. using finite state machines to generate test cases [109, 110]. These models generally have to be constructed manually, but automatic approaches, e.g. GUI ripping proposed by Memon [111], also exist. The benefit with GUI ripping is that it mitigates the extensive costs related to model creation. Costs that originate in the complexities of developing a suitable model. The limitation of this approach is that it is dependent on the SUT implementation, e.g. development language.

The area of GUI interaction based testing and automation is therefore quite broad but still limited in regards of empirical studies in real-world contexts with industrial grade software systems. R&R tools have been compared [106] and evaluated in industry, for both system- and acceptance-test automation, but, to our best knowledge, it is only our own work that evaluates VGT in an industrial context [65]. Our previous work is however limited since it was conducted only for a small set of real-world test cases and since the VGT automation was performed by researchers rather than practitioners. Hence, the body of knowledge on VGT, to the authors best knowledge, lacks industrial case studies that report on the real-world use of the technique.

Most research on GUI based testing focuses on system testing. However, acceptance testing is an equally important, valid and plausible test aspect to consider, i.e. tests where requirements conformity is validated through end user scenarios performed regularly on the SUT [38]. Scenario based acceptance tests do however distinguish themselves from system tests by including more end user specific interaction information, i.e. how the system will be used in the end users' domain. Automated acceptance testing has also been a subject of much research, which has resulted in both frameworks and tools, including research into GUI interaction tools [31]. However, to the authors' best knowledge, only our previous work has considered the subject of using VGT for acceptance testing.

3.3 Research methodology

This section will present the company where the VGT transition was performed as well as the research methodology used to collect data during the case study.

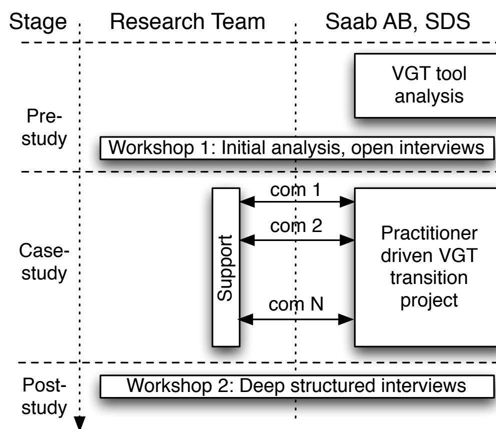

Figure 3.1: *Overview of the case study, including the two performed workshops and the continuous, yet discrete, communication between the company and the research team. Note that the academic support effort is considerably smaller than the VGT transition effort.*

3.3.1 Research site

The case study presented in this paper was conducted in collaboration with, and at, the Swedish company Saab AB, subdivision SDS, in the continuation of this paper referred to as Saab. The study was conducted at the company because they had taken the initial steps towards transitioning into VGT to automate their current manual testing, which presented an opportunity to collect data to bridge the current gap regarding VGT's real world applicability. Figure 3.1 visualizes the stages of the case study, which will be presented in more detail in the following section based on the guidelines for reporting case studies presented by Runeson and Höst, 2009 [17].

Saab develops military control systems for the Swedish military service provider on behalf of the Swedish military forces. The system is, when deployed in the field, distributed between several mobile nodes and provides the ability to map the position of friendly and hostile forces on the battlefield and share this information among the nodes. Hence, the core functionality of the system relies on a map visualization, provided by a map engine, which allows the user to place symbols representing military units onto the map. Due to the system's intended use it is considered both safety and mission critical. In addition, the system is developed for a touchscreen monitor for use while the node is in motion, i.e. buttons and other graphical GUI objects are larger than a conventional desktop application to mitigate faulty system interaction when used in rough terrain. The system is both developed and maintained by the company, with a development team that is independent from the testing team. In addition, the system has a very large and complex requirements specification aligned with 40 test specifications built from roughly 4000 use cases which has an estimated manual execution time of 60 man-weeks (2400 man-hours).

3.3.2 Research process

The case study consisted of three stages, shown in the leftmost column (named 'Stage') in Figure 3.1. The first stage was explorative in nature, the second sought to improve and support the VGT transition and the third was descriptive in nature. In the first stage, the row named 'Pre-study' in Figure 3.1, a workshop was conducted with the goal of collecting information about the company's goals with the VGT transition, their manual test practices, the SUT, etc. This information was collected using unstructured open interviews with the testers that were driving the VGT transition at the company. Unstructured open interviews were chosen because very little was known about the company at this stage of the study. In addition, several documents were acquired that could provide further information about the manual test suite and the SUT.

In the second stage of the case study, which was four calendar months, a communication process was followed to allow the testers driving the VGT transition and the research team to exchange information on a regular basis, i.e. the row named 'Case study' in Figure 3.1. The communication process was put in place for two reasons. First because the project was to be driven by the testers at the company rather than the research team; the latter deliberately distanced themselves from the project in order for all collected data to genuinely portray VGT's use in the real world. The second reason was out of necessity due to the physical distance, i.e. 500 kilometers, between the research team's location and the company. The information exchange took place more often at the start of the project, at least once each week, since the research team had deeper understanding of VGT than the testers, i.e. the research team could provide the testers with expert support. This support included information of how to improve the VGT test suite that was being constructed but also suggestions of how to document the test suite and solutions to specific, low-level, problems that the testers had run into. In cases where the research team did not already have a feasible solution to a problem, the research team instead aided in the information acquisition to help the testers develop a solution. Further into the project, the information exchange became less frequent with telephone or mail communication roughly twice each month. During these discrete instances, challenges, limitations and solutions were discussed as well as the progress of the VGT transition. In addition, cost and time metrics were collected from the testers. Hence, the role of the research team in this stage of the project was two-fold. First to provide support for the VGT transition project, and second to acquire empirical data regarding the VGT transition from the testers.

In the third stage of the study, which aimed to portray the project and its outcome, a second workshop was held on site at the company, during which two structured deep interviews were held with the driving testers, shown in the row named 'Post-study' in Figure 3.1. Additionally, at this point of the project, an additional tester had joined the transition project who could provide a new perspective and further information about the transition and usage of VGT. The purpose of the interviews was to verify previously collected data, get a deeper understanding of the transition project as well as to collect further data on challenges, limitations and solutions that had been identified. Both

of the interviews were recorded and conducted using the same set of questions in order to raise the internal validity of the answers [17]. 71 questions were prepared for the interviews, 67 with the purpose of eliciting and validating previously collected information and 4 attitude questions aimed at capturing the testers views on VGT, post project. More specifically, the four questions were,

Q1: Does VGT work? Yes/No, why?

Q2: Is VGT an alternative or only a complement to manual testing?

Q3: Which are the largest problems with VGT?

Q4: What must be changed in the VGT tool, Sikuli, to make it more applicable?

In all of the questions, VGT refers to VGT performed with Sikuli [54], since Sikuli was the VGT tool that was used during the project. After the interviews, the recordings were transcribed in order to make the information more accessible. In addition, the answers were analyzed and compared among the respondents, i.e. the driving testers, to ensure that there were no inconsistencies in the factual data. The analysis showed that the respondents had answered the majority of the questions the same, including all attitude questions, but that they had complementing views on the attitude questions, i.e. what was the largest issue with working with VGT, etc.

3.4 Results and Analysis

The following section presents the results, and analysis of the results, divided according to the three stages of the VGT transition project, i.e. pre-transition (pre-study), during the transition (case-study) and post-transition (post-study) to VGT.

3.4.1 Pre-transition

The VGT transition at Saab was initiated out of necessity to shorten the time spent on manual testing. For each release, every six months to one year, the SUT went through extensive regression testing where a selected subset of the SUT's test cases were manually performed. Each regression test session had a budget of four to six weeks of man-hours. The test cases were documented in 40 test suites, referred to as acceptance test descriptions (ATD). Each ATD consisted of a considerable set of use cases (UC), e.g. roughly 100, which each defined valid SUT input and the expected output. On a meta level these UCs were linked together into test chains that defined the test case scenarios, as exemplified in Figure 3.2. A test case was defined as a test path through a test chain that could be either linear, or contain branches, where a set of UCs, UC1 and UC2 (Top left of Figure 3.2), were first executed to set up the SUT in a specific state. The set up was then followed by the execution of one of a set of optional UCs, UC3A-C (Middle of Figure 3.2) to create a test path. Test paths could also have varying length, as exemplified in the figure where UC3A (Middle left in Figure 3.2) is followed by UC3AA (Bottom of Figure 3.2) while

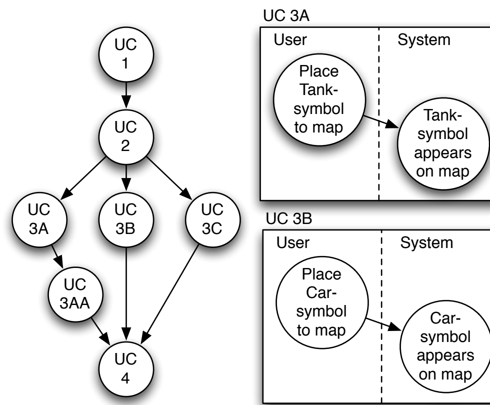

Figure 3.2: *Example of a acceptance test description (ATD) test chain (to the left) constructed from a set of ATD use cases (to the right). In the example the test chain contains three unique test-paths, i.e. test cases, that were, prior to the VGT transition, executed manually.* ***UC*** *- Use case.*

the other two branches (UC3B and UC3C) lack following UCs. Hence, each test chain could contained a set of branching test-paths, i.e. test cases, defined by either common or unique UCs. The modular architecture of the manual test cases provided a lot of flexibility but was also considered tedious since some test chains required a lot of setup while only performing a small/short test thereafter.

The manual test period, four to six weeks, for the SUT, was then followed by a factory acceptance test (FAT) with the customer, executed over an additional two to three weeks, to validate the system, i.e. six to ten weeks of testing in total. However, a FAT would only be initiated if the manual tests had been executed successfully. Hence, transitioning to VGT from manual testing would constitute a large gain for the company in terms of development time, cost and potentially raised quality, since a larger subset of test cases from the ATDs could be executed faster and at higher frequency [65]. Raising test frequency was also important since manual testing was the only means of testing the system, i.e. no other tests existed for regression testing purposes such as automated unit tests, etc.

Three VGT tools were evaluated for the project, i.e. EggPlant, Squish and Sikuli, to find one suitable for the VGT transition. A brief overview of the results of the evaluation is given in Table 3.1. The primary success factors during the evaluation, which took six man-weeks, were tool cost and script language ease of use. Each tool was evaluated based on its static properties as well as through ad hoc scripting and automation of actual use cases from the ATDs. In addition, the evaluation took into consideration the research teams' previous work, i.e. comparison of different VGT tools [65].

The result of the evaluation was that EggPlant was a mature and suitable tool but that it was very expensive and that the tool's scripting language was a limitation, i.e. it had a high learning curve and did not suit the modular design of the tests that the testers were aiming for. Squish, used by other departments

Tool	Advantages	Disadvantages
EggPlant	VNC support, Mature product, Powerful	High cost, Script language limitations
Squish	Reference based, fast	Limited thread based interaction, inability to work with the map
Sikuli	Open source (free), flexible, Python scripting language	Volatile IDE, lacks test suite support

Table 3.1: *Summary of advantages and disadvantages of the VGT tools evaluated during the VGT transition project.*

at Saab, was not suitable either since it performed GUI based testing through manipulation of execution threads in the application. However, the SUT was running roughly 40 threads at a time, spread over different system components, which limited Squish ability to interact with the SUT. Additionally, the tool was unable to identify objects placed on the map, due to its limited image recognition capabilities, which was a key feature of the SUT that the VGT tool had to be able to cope with in order to be applicable. Lastly, Sikuli was evaluated and found to be a feasible option, partly because the tool is open source, and thereby carries no up front cost, but mostly because of the tool's scripting language which is based on Python. Python was considered valuable since it has a familiar syntax, i.e. common to most imperative and object-oriented programming languages, and because Python provides the capabilities of an object-oriented programming language. The main limitation with Sikuli, that was identified at this stage of the project, was that the tool did not have built in support for either development or management of test suites. However, thanks to the power of the tool's scripting language this was considered a minor obstacle since a custom solution could easily be developed by importing and extending existing testing and test suite libraries for Python. Another problem that was identified was that Sikuli did not have any built in virtual network connection (VNC) support, required to test the SUT's distributed functionality. However, by pairing Sikuli with a third part VNC client-server application, this issue was also easily solved.

3.4.2 During transition

The VGT transition took place during roughly four calendar months, during which three representative ATDs were fully implemented into a VGT test suite. Representativeness was measured by the ATD's complexity, where two of the chosen ATDs were considered more complex than the average 40 ATDs, whilst the third was equal in complexity to the remaining ATDs. The VGT test suite architecture, visualized in Figure 3.3, consisted of two main parts. First, a main script for each ATD that imported all the automated ATD test cases, i.e. the test chains built from use cases. The second part was the test cases themselves which were executed by the main script according to the numerous test paths in each test chain. This architecture was required since Sikuli does

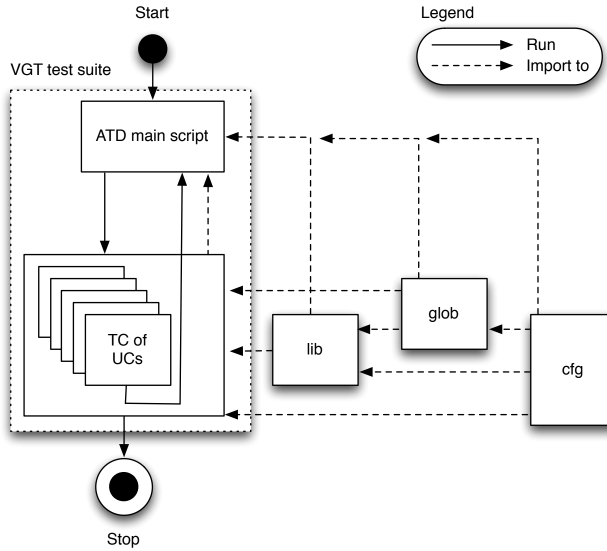

Figure 3.3: *VGT test suite architecture.* ***TC*** *- Test case,* ***UCs*** *- Use cases.*

not, as mentioned, provide any support for either development or management of test suites. The VGT test suite was also developed using external libraries, 'lib' in Figure 3.3. One of these libraries was a Python library for formatting and producing output. Output that could be viewed graphically through any web browser, i.e. the result of each test case was visualized as passed or failed in a table. Additionally, a Java library for taking screenshots was incorporated in the VGT test suite. The screenshots provided additional value to the result output by capturing the state of the system when a bug was identified, i.e. the faulty state of the GUI was captured for further analysis and for manual recreation of the bug. According to the testers, this functionality made it easier to explain, and present, the faults they encountered to the developers, thereby quickening SUT maintenance time. In addition, all global variables used in the scripts were placed in its own library called 'glob', whilst the 'cfg' library included all external paths, i.e. paths to where to save log files, find the external libraries, etc.

After automation of the three ATDs, the testers compared the VGT test suite's execution time against the manual test suite execution time. Results showed an estimated speed up of a factor 16, from two work days (16 hours) to 1 hour for the two complex ATDs and from 1 day to 30 minutes for the third. Hence, the automation constituted a huge gain in test case execution time with no reported detrimental effects on bug finding ability, i.e. *all bugs in the system that were identified using manual test practices could also be identified using the VGT test suite.* In addition, due to the quicker execution speed, the automated ATDs could be run several times in sequence. The iterative test suite execution placed the SUT in states that the manual test cases did not cover. Consequently, three new faults were uncovered that had not been identified earlier with the manual testing. In addition, these bugs were automatically captured and recorded by the screenshot capabilities of the

VGT test suite which made them simpler to present, recreate and motivate as faulty behavior to the developers. However, even with the much higher execution speed, the testers reported that they were often asked, *"Doesn't it execute quicker than this?"*. The simple answer, as reported by one of the testers, is, "Sikuli, or VGT, is limited by the speed of the SUT", i.e. the VGT test suite cannot run faster than the reaction speed of the SUT's GUI. Consequently, the scripts often had to be slowed down, using delays, in order to synchronize them with SUT loading times to ensure that the SUT's GUI was ready for new input before the script continued its execution.

During development, attempts were made to integrate the VGT test suite into the SUT's build system, i.e. to allow completely automatic system regression testing after each new build. However, since the VGT test suite required manual setup and some configuration before execution, such a scheme was never implemented due to time constraints. Instead, the VGT test suite was run on an ad hoc basis, i.e. not periodically, but with much higher frequency than the previous manual testing. The higher frequency regression testing was reported as most beneficial for the development of the SUT since it provided the developers with quicker feedback.

3.4.2.1 VGT test suite maintenance for improvement

To ensure validity of the automated test scripts, they were developed as a 1-to-1 mapping of the manual test cases, i.e. the manual tests were used as a specification for the automated scripts. However, later during the project, the VGT test suite was subject to maintenance. The maintenance done to the test case scenarios included, but was not restricted to, modification of the order of script operations, in order to provide smoother and quicker test case execution, and further modularization to facilitate strategic reuse. Hence, breaking the 1-to-1 mapping in some of the test cases. However, the purpose of each automated test case, i.e. the functionality the test case aimed to verify in the SUT, was kept the same. Consequently, a conclusion can be drawn that strict automation, i.e. 1-to-1 mapping, of the manual test specification may not necessarily be the best automation approach. Rather, the specification should only be used to specify what to test in the SUT, not necessarily how. The reason is because with automatic testing you can, and often want, to improve the test execution speed as much as possible, which can be done by grouping certain actions together. In contrast, manual test scenarios need to be unambiguous and test actions defined logically to have high quality [115], which isn't necessarily the fastest. Hence, the quality of a VGT script is greatly affected by how it is designed and implemented, i.e. narrowing the gap between testing and traditional software development.

The performed refactoring of the VGT test suite was required since this project was conducted under continuous time pressure, with project managers expecting quick results. This pressure resulted in, as presented by the testers, development of the first possible solution for certain problems which necessarily wasn't always the best solution in terms of script quality, performance, reusability, etc. Additional refactoring was also required due to the testers inexperience of using Sikuli at the start of the project. Among the refactoring that was made, in order to improve maintenance of the scripts, all global

variables were moved to a common namespace, i.e. 'glob', as shown in Figure 3.3. Hence, all variables were clustered in one library and then, together with the libraries, 'lib' and 'cfg', imported to all scripts that required them.

During the VGT test suite maintenance, the testers observed that it was easier to maintain the scripts that they had written last since they had a clearer memory of what the scripts did. Additionally, they reported that whilst maintenance of their own code was almost as quick as writing code from scratch, maintenance of scripts written by the other tester took considerably longer. One solution to mitigate these problems would have been a common coding standard of how to name variables, write loops and branches, etc. This problem, once again, illustrates how VGT, using Sikuli, in many respects has more in common with traditional software development than testing. However, as reported by the testers and in contrast to traditional development, the maintenance work was made easier by the scenario based structure of the scripts and the intuitiveness provided by inclusion of images in the scripts, a feature provided by Sikuli's IDE. It was perceived by the testers that pure Python code would have been more difficult to maintain; the in-script images simplified understanding and remembrance.

3.4.2.2 VGT test suite maintenance required due to SUT change

Three calendar months into the VGT transition project a huge change was made to the SUT which included replacement of the map engine. Since the map engine was part of the core functionality of the SUT this change also affected the VGT test suite, i.e. causing 85-90 percent of the scripts to fail and thereby require some kind of maintenance, which included changing 5-30 percent of the images in every maintained script. The maintenance effort required to get the VGT test suite working completely again took roughly three man-weeks (240 man-hours) of work, which is to be related to the VGT test suite development time of three man-months (1032 man-hours). Hence, the estimated maintenance time of the entire VGT test suite, all 100 percent of the test scripts, would be 25.8 percent of the development time, i.e. 266 man-hours, which can be compared to the manual test budget of 480 man-hours per SUT development iteration. Note, the 4-6 week manual execution time, 120 hours, is with two testers. Consequently, the estimated development time of all 40 ATDs would be 13760 man-hours (7.6 man-years) and assuming all of the tests broke, the maintenance time would be 3550 man-hours, equal to roughly 21 man-months of continuous work or equivalent to the budge of 7 iterations of manual testing, i.e. roughly 3.5 years. However, the time required to execute all of the 40 ATDs manually is estimated to 2400 hours. Hence, assuming that none of the tests required maintenance and the complete VGT test suite (40 ATDs) was executed continuously, i.e. 24 hours a day, the ROI for the entire development would be positive after roughly 8 days (199 hours), i.e. after executing all the 40 automated ATD's 6 times. Additionally, for the three ATDs that were automated in the project, a positive ROI would be reached after 13 executions, i.e. after 32.5 hours of continuous execution, which is less than the time of the manual ATD execution, i.e. 80 man-hours.

These numbers, summarized in Table 3.2, do however not reflect the manual testing that is performed during the VGT test script development, required

Artefact	**Dev. time**	**Maintenance of VGT test suite**	**Man. exe. time**	**Positive ROI reached after**
VGT test suite (Project)	1032mh	266mh	80mh	13 VGT test suite executions
Entire test suite (Estimated)	13760mh	3550mh	2400mh	6 VGT test suite executions

Table 3.2: *Summary of development-, maintenance- and manual execution times (man-hours) and return on investment (ROI) (VGT test suite dev. time / manual exe. time) data acquired from the VGT transition project.* ***mh*** *- man-hour,* ***h*** *- hour*

to validate test script conformance to the manual test specifications. Furthermore, the numbers do not take into account aspects such as the number of faults found during the test execution, i.e. quality gained from quicker feedback to the SUT developers and other benefits provided by the VGT test suite, e.g. identification of previously unknown bugs. With these aspects taken into account, the driving testers estimated that the currently achieved ROI of the VGT transition was neither positive or negative. Hence, their perception is that all future regression testing performed with the VGT test suite will provide positive ROI for the company. However, the numbers also show that it would be unfeasible to automate all the 40 ATDs since it would take 7.5 man-years. Hence, an important conclusion is therefore that a company may have to prioritize or be selective in which manual test suites they decide to automate. Furthermore, as described by the testers, VGT primarily solves cost and speed problems rather than raising quality. The higher test frequency can help identify bugs faster, but bugs are only found if covered by the test scenarios.

The testers encountered a set of additional problems during the VGT transition, which have been summarized in Table 3.3. The main problem was the volatility and instability of the VGT tool, i.e. Sikuli. Sikuli is still a release candidate, i.e. not a finished product, and therefore suffers from some lingering bugs. These bugs affect the stability of the tool's IDE that is prone to failure in certain instances, e.g. if the execution thread of a script is manually terminated, or if the tool is terminated with an unsaved script, etc. The solution to solve these problems has been to only use Sikuli's IDE for script development and instead run the developed VGT test suite from the command line, which was found to greatly improve stability.

The single largest problem, as described by the developers, was however the failure rate of Sikuli's image recognition algorithm, which was not improved by running the scripts from the command line. Estimates done by the testers indicate that the VGT test suite only had a success rate of 70 percent. This low success rate has been established by the testers to be due to the use of

VNC. The VNC server-viewer application is used to run test cases that are distributed over several physical computers. However, not all of the test cases require the VNC connection and when these tests were executed against the SUT, without VNC, the testers observed a close to 100 percent success rate, even when the VGT test suite was left to its own devices for over 24 hours. Consequently, the solution that was employed, during the pre-transition stage of the project, to allow Sikuli to test the distributed system, also proved to be the largest problem for the stability of the scripts. The cause of the problem has not yet been verified but the hypothesis is that the problem is related to network latency, causing the remote images sent from the VNC server to the VNC viewer to be distorted, causing the image recognition algorithm to fail.

Additional problems caused by the VNC solution relates to the mouse pointer. Sikuli, when executed locally, disregards the mouse pointer, i.e. removes it from the screen, when it's performing the image recognition. However, when executed over VNC the mouse pointer cannot be removed and if placed in the wrong position, e.g. in front of the sought button, it causes the image recognition to fail. The problem can easily be mitigated by adding operations in the script to continuously move the mouse pointer to a safe location. However, this solution is inconvenient and adds unnecessary code and execution time to the scripts. Additionally, as reported by the testers, it adds frustration to the script development.

Yet, even though there were many problems, challenges and limitations that hindered the VGT transition, the testers still claim that they had not encountered anything that they could not automate using Sikuli. Additionally, the testers experienced that the development itself contributed to raising the quality of the SUT since it required them to perform the test cases manually several times to obtain a greater knowledge of how to automate them. Hence, the development work itself helped uncover several faults in the SUT. Faults that could later also be identified automatically by the VGT test suite.

3.4.3 Post-transition

After the VGT transition was completed, a second workshop was held on site at the company during which structured interviews were performed with the testers driving the project. The purpose of the interviews was primarily to verify previously collected information but also to capture the testers views on if VGT is viable for system- and acceptance-testing in industry.

During the interviews, four attitude questions were asked, presented in Section 3.3 and summarized in Table 3.4. For the first question, does VGT work, the interviewees were clear that it did. Two motivations stated by one of the testers was, *"It is such a good way to quickly run through and make sure that everything still works and you can use it on any system"*. An additional motivation from another tester was, *"VGT is the only thing that works on our system"*. Hence, VGT is perceived not to be bound to any specific implementation language, API, etc., and its image recognition capabilities therefore allows it not only to interact with one application at a time, but seamlessly interact with different applications at once.

For the second question, when asked if VGT is a complement or a replacement for manual testing, the testers stated that it is a complement, *"It's part of*

Title	Problem	Solution
VNC	VNC has negative effects on the image recognitions ability to identify GUI graphics	Minimize use of VNC if possible, use high-quality VNC application, use EggPlant
Maintenance	Understanding other developers scripts can be problematic even with the scenario based structure of the scripts	Enforce coding standards to raise understandability and readability of the scripts
1-to-1 mapping	1-to-1 mapping between manual and automated tests is not always possible or favorable	Modularization of test scripts can increase test execution speed and reusability. Hence, a 1-to-1 mapping should be strived for only if it does not have detrimental effects on test quality.
Sikuli IDE volatility	Sikuli is not a finished product and therefore cause the Sikuli IDE to fail unexpectedly	Use IDE only for script development but execute scripts from command-line
Lack of documentation	Sikuli's API is poorly documented	Ensure internet connectivity to make it possible to look up solutions and other information online.
Image recognition	Many problems were identified with Sikuli's image recognition, e.g. spontaneous inability to find images, click operations performed next to intended location, etc.	No one solution was identified, but potential solutions include fine-tuning the scripts, better selection of images, running scripts locally without VNC, etc.

Table 3.3: *Summary of problems and solutions identified during the VGT transition project.*

Nr	Question	Answer
1	Does VGT work? Yes/No, why?	Yes, only technique the testers have identified capable of automating their manual tests.
2	Is VGT an alternative or only a complement to manual testing?	complement, since it can only find faults covered by the scripted scenarios.
3	Which is the largest problem with VGT?	The volatility of the tool and the image recognition.
4	What must be changed in the VGT tool, Sikuli, to make it more applicable?	Support for testing of distributed systems, e.g. through VNC.

Table 3.4: *Summary of the driving testers' responses to the four attitude questions asked during the second workshop.*

the test palette". Based on their perception, VGT may work as a replacement for smaller systems, but for large and complex systems it is neither suitable or plausible that this could be achieved. The reason is because it is improbable that test scenarios can be devised that cover all states of a large systems, which is equally unlikely for manual scenario based test cases. Instead, manual exploratory testing should be used to uncover new faults.

For the third question, what is the biggest problem with VGT, one of the testers stated, *"I don't see any problems with it, but we need to get around the fact that it does not always work and that we always don't know why."*, referring to Sikuli's volatility. Another tester answered, *"The image recognition comes with an inherent uncertainty"*, i.e. fragility to unexpected SUT behavior, etc. However, the testers had a pragmatic approach to these issues and stated, *"Sikuli is a program, it's also a system and systems have faults"*. Hence, they had accepted the tools limitations but also identified that most of these limitations could be mitigated through structured script development, redundancies in the scripts and other failure mitigation practices.

Finally, when asked what can be improved with the VGT tool, the testers answered that the reliability of the tool should be increased or at least a study should be conducted that can explain why the image recognition works in some cases, for some images, and not for others. Additionally, the tool documentation needs to be improved and since one of the largest issues during the VGT transition was found to be how the tool interacted with VNC, Sikuli should be fitted with VNC capability, similar to EggPlant. As stated by the developers, *"EggPlant was much more stable with VNC. We have not managed to make Sikuli as stable."*.

Due to the success of the transition project, i.e. identification of previously unknown faults in the SUT and the perceived cost-effectiveness of the technique, the use of VGT has also been accepted by the customer as a complement to the manual testing. Additionally, because of the success, the company plans to continue the automation of more ATDs and also develop a new VGT test suite to test all basic functionality of the SUT. This new VGT test

suite will not be based on the manual ATDs but rather on domain knowledge about the intended low-level functionality of the SUT. The testers at Saab have also started looking at the possibility of creating an automatic thread-based exploratory test (TBET) based VGT application. TBET, a refinement of exploratory testing [116], is executed by following one or several execution threads, scenarios, through the SUT to find faults, and also their causes. However, no actual implementation had been conducted on such a solution at the time of the project.

Hence, it can be concluded that even though VGT has its limitations, challenges and problems, it is still a viable and applicable technique for industrial use when performed by practitioners. This conclusion is strengthened by the impact that the transition project has had within the Saab corporation where more Saab companies have started working with the technique. Even though, as reported by the testers, there are naysayers claiming that *"Automation did not work 25 years ago and therefore it won't work now."*. However, in this paper we have presented information that contradicts the naysayers claims, e.g. feasible development and maintenance costs, raised fault finding ability.

3.5 Discussion

The data collected during the industrial case study shows that the transition to VGT was both successful and of benefit to Saab, benefits summarized in Table 3.5. Firstly, the execution speed of the company's previously manual tests was greatly improved that allows for greater test frequency and thereby faster feedback to the developers, i.e. from months to hours. Secondly, and perhaps more importantly, the automated tests did not just identify all the faults found by the manual tests, but also previously unknown faults. Consequently, this report provides support that VGT does not just lower testing costs, but can also helps raise software quality. However, as also reported, the transition cost of several large manual test suites can be extensive, so a cost-benefit prioritization model of what test suites to automate should be developed, which is a subject of future work. Thirdly, the return on investment (ROI) of transitioning to the technique, i.e. automating the manual tests, was perceived by the driving testers to become positive after only one iteration of SUT development. A claim supported by our previous research [65], which came to the same conclusion at another Saab company. Additional support comes from the fact that the manual tests are continuously performed during VGT transition to ensure script validity, i.e. not taking time away from the normal manual testing, and the benefit of faster fault identification due to raised test frequency. Manual testing cost increases linearly with each development iteration, but VGT only has an initial cost for developing the automated test suites after which the cost of executing the scripts is constant. Hence, due to the execution speed of a VGT test suite, the number of executions required to reach a positive ROI can be performed quickly, as shown in Table 3.2.

Additionally, as shown in Table 3.5, other improvements were identified that are of benefit for future use of VGT and compared with previous GUI testing techniques, e.g. record and replay (R&R). Firstly, results show that the maintenance costs of a VGT test suite are not excessive, i.e. 25 percent

Description	Past	Current	Benefit (versus manual testing) or improvement
ATD execution time	1-2 days per ATD (60 man-weeks for all ATDs), manually	0.5-1 hour per auto. ATD (Estimated 33 hours for all automated ATDs)	Test execution 16 times faster, higher test frequency, quicker feedback to developers
Fault finding	-	3, previously unknown, faults found	VGT provides greater fault identification ability, higher system quality
Test ROI	Linear cost (Manually)	Constant cost after 1 iteration (Automatic)	Positive ROI after one iteration, feasible development cost
Script maintenance cost	Unfeasible in the worst case for previous GUI test techniques (record and replay) [12]	~25% of the development cost of the VGT test suite (Saab project, with Sikuli)	Maintenance cost perceived feasible
Sikuli executed over VNC	~70% success rate with VNC	100% success rate without VNC	Sikuli stable when executed locally

Table 3.5: *Summary of quantitative benefits and improvements identified during the VGT transition project.*

of the development cost. In addition, the script refactoring was generally contained to parts of or specific scripts, which should be compared to the required maintenance of previous techniques, e.g. R&R, where entire test suites were rendered useless due to SUT change. Consequently, the black box nature of VGT, due to the image recognition, makes changes to the SUT maintainable. However, the collected data is not enough to draw a definitive conclusion that the maintenance costs of VGT scripts are feasible for industrial use; more research is needed on this in the future.

Secondly, Table 3.5 presents data regarding the stability of VGT when used together with a virtual network connection (VNC). VNC was used during the project because the SUT was distributed over several computers. However, this pairing was recognized as a large problem since it lowered the success rate of the automated test suite, when it should have succeeded, to roughly 70 percent, i.e. due to image recognition failures. The VNC problem was identified by running a subset of test scripts, which could be run locally, against the SUT that resulted in a success rate of 100 percent, even when the tests were rerun continuously for 24 hours. Hence, Sikuli's image recognition was not the source of the problem, but rather it was the third party VNC application, mitigated by local VGT test script execution. However, since the system was distributed over several computers, i.e. nodes, this solution instead limited which test cases could be executed. Hence, this was not identified as a benefit but rather an improvement of how to use Sikuli to raise test suite stability. Consequently, either a better VNC application has to be obtained or VNC should be integrated into Sikuli as already available in the VGT tool EggPlant, which was perceived by the testers be much more stabile in this regard. However, EggPlant, as reported by the testers, had other limitations, e.g. a high cost and, what they considered, an unintuitive and more restricting scripting language. Consequently, existing VGT tools suffer from important, but different limitations, that makes it likely that manual test execution will still have to complement automated testing. However, the testers' common view is that VGT both works and provides substantial value to the company, even given the tools' limitations.

The testers also identified other less quantifiable benefits with VGT during the project. One benefit being the techniques flexibility and ability to work with any application regardless of implementation language or even platform, i.e. web, desktop, mobile, etc. This flexibility allows VGT to interact with the SUT whilst also interacting with SUT related simulators, written in other programming languages, or even the operating system if required. This is a specific benefit of VGT that might or might not be present with other similar testing tools, such as R&R or GUI testing techniques that are specific to the GUI library in use. In addition, the VGT tool that was used, i.e. Sikuli, uses Python as a scripting language that provides the user with all of the properties of a lightweight, object-oriented programming language. These properties presents new interesting opportunities for automated testing but also new problems. Since the scripts follow the rules of traditional software development they are also subject to the same types of faults, i.e. if implemented incorrectly they can contain bugs. Consequently, an inherent risk with complex scripts is that they report type 2 errors, i.e. false negatives, due to the scripts themselves being faulty. Hence, the question becomes, how do

you verify the tests? Verification of scenario based scripts that strictly follow a manual test description can perceivably be done through comparison with the outcome of the manual tests. However, for more advanced VGT-based test applications a more complex verification technique might be required, e.g. based on oracles or properties, or other state-of-art techniques, to ensure that all faults in the SUT are identified.

3.5.1 Threats to validity

The main threat to the validity of this study is that it only presents results from one VGT transition project at one company. Hence, the results may have low external validity for other companies and domains [17]. In addition, since no structured data collection process could be performed by the driving testers during the project, due to resource constraints on the company's end, quantitative metrics were only sparsely collected. The risk that very little quantitative data would be available from the project was identified already before the case study started and originates in the fact that this project was performed in a real-world context with real-world time constraints. Consequently, the results presented in this work are primarily based on data collected through interviews and are therefore mostly qualitative in nature. Further work is therefore required in more companies to provide additional support regarding the real-world applicability of VGT. Another threat is that the driving testers at the company might have been biased, i.e. wanting the transition to be successful. However, based on their thorough descriptions of faults, limitations and problems, this threat is considered minor.

3.6 Conclusion

In this paper we present an industrial case study from a successful visual GUI testing (VGT) transition project, performed by practitioners, at the company Saab AB, subdivision SDS. Additionally, problems, limitations and solutions that were identified during the project are presented. Furthermore, support is given that the maintenance costs of a VGT test suite, developed in Sikuli, are not excessive, i.e. in this project 25.8 percent of the VGT test suite development cost.

In previous work we have shown the industrial applicability of VGT, but in a smaller transitioning project driven by researchers with expert knowledge of the technique. The more extensive transitioning project presented in this paper was instead initiated from industry, and originated in the business need to shorten the execution time of manual regression testing. The main limitation of the VGT tool, Sikuli, used during the project, was its unpredictability, e.g. uncertain image recognition outcome and tool IDE instability, which was partly mitigated through local test suite execution via the command line. The benefits of VGT were reported to be the technique's flexibility to work with any application, greatly improved test execution speed (16 times faster than manual tests) and ability to identify all faults found by the previous manual tests. Furthermore, the VGT test suite could identify previously unknown faults, due to increased test execution speed that allowed the tests to be run several times in sequence. Results also showed that the VGT transition cost,

of three automated acceptance test descriptions (ATD), was feasible, but that VGT transition of all of the company's 40 ATDs would take 7.5 man-years of work, i.e. prioritization of the ATD transition will be required. However, the practitioners perception was still that the developed VGT test suite was beneficial and will provide the company with positive return on investment for all future use. Hence, even though there were problems and limitations, the practitioners' perceptions, and collected data, show that VGT is a beneficial and feasible technique for industrial system test automation.

Chapter 4

Paper C: Challenges, problems and limitations

Visual GUI Testing in Practice: Challenges, Problems and Limitations

E. Alégroth, R. Feldt, L. Ryrholm

Published in the Empirical Software Engineering Journal, 2014.

Abstract

In today's software development industry, high-level tests such as system and acceptance tests are mostly performed with manual practices that are often costly, tedious and error prone. Test automation has been proposed to solve these problems but most automation techniques approach testing from a lower level of system abstraction. Their suitability for high-level tests has been questioned. High-level test automation techniques such as record and replay exist, but studies suggest that these techniques suffer from limitations, e.g. sensitivity to GUI layout or code changes, system implementation dependencies, etc.

Visual GUI Testing (VGT) is an emerging technique in industrial practice that aims to overcome many of the limitations experienced with previous test automation techniques. The core of VGT is image recognition applied to analyze and interact with the bitmap layer of a system's front end. By coupling image recognition with test scripts, VGT tools can emulate end user behavior on almost any GUI-based system, regardless of implementation language, operating system or platform. However, VGT is not without its own challenges, problems and limitations (CPLs) but, like for many other automated test techniques, there is a lack of empirically grounded knowledge of these CPLs and how they impact industrial applicability. There is also a lack of information on the cost of applying this type of test automation in industry.

This paper reports an empirical, multi-unit case study performed at two Swedish companies that develop safety-critical software, while they transitioned their manual system test cases into tests automated with VGT. One complete test suite, consisting of 33 test cases, was automated in one of the projects and three test suites in the other, exact number of test cases is uncertain because of the structure of the manual test cases. The results from the study participants showed that the transitioned test cases could find defects in the tested systems and that no manual test cases were found that could not be automated. During these transition projects, a total of 58 different CPLs were identified and then categorized into 29 groups. These groups are presented and their implications for the transition to and use of VGT in industry is analyzed. In addition, four high-level practices are presented that were identified during the study, which would address about half of the identified CPLs. Furthermore, collected metrics on cost and return on investment of the VGT transition are reported together with information about the VGT suites' defect finding ability. A total of nine defects were identified by the automated tests or during the transition, of which 5 were unknown to testers with extensive experience from using the manual test suites. The main conclusion from this study is that even though there are many challenges in transitioning to automated testing based on VGT, the technique is still valuable, flexible and considered cost-effective by the industrial practitioners.

4.1 Introduction

Currently available automation techniques for high-level testing, i.e. system and acceptance testing, leave more to be desired in terms of cost efficiency and flexibility, leaving a need for more empirical research into test automation [3, 4, 12, 13, 16, 52, 97, 117]. This need for automation support is one factor why manual testing is persistently used in industrial practice even though these practices are considered costly, tedious and error prone [2–7, 118].

Manual testing has long been the main approach for testing and quality assurance in the software industry and in recent years there has been a renewed interest in approaches such as exploratory testing that focus on the creativity and experience of the tester [116]. The more traditional manual tests are often pre-defined sets of steps performed on a high level of system abstraction to verify or validate system conformance to its requirement specification, i.e. system tests, or that the system behaves as expected in its intended domain, i.e. acceptance tests [25, 38, 102, 103]. However, software is prone to change and therefore requires regression testing [8,9], which can lead to excessive costs since testers continuously need to re-test. Manually having to repeatedly follow the same test descriptions and look for the same problems is also tedious and error prone.

To mitigate these problems, whilst retaining or increasing the end quality of the software being tested, automated testing has been proposed as a key solution [7]. Many automated test techniques approach testing from lower levels of system abstraction, e.g. unit tests [11, 14, 32], which make them a powerful tool for finding component and function level faults. However, the use of these techniques for high-level testing has been questioned since empirical evidence suggests that high-level tests based on low-level test techniques become complex, costly and hard to maintain [3, 16].

Thus, a considerable body of work has been devoted to high-level test automation, resulting in techniques such as coordinate- and widget/component-based Record and Replay with a plethora of tools such as Selenium [55], JFCUnit [119], TestComplete [120], etc. The tools record the coordinates or properties of GUI-components during manual user interaction with the system's GUI. These recordings can then be played back to emulate user interaction and assert system correctness automatically during regression testing. However, empirical studies has identified limitations with these techniques. They are typically sensitivity to GUI layout or code change and tools dependend on the specifics of the system implementation, etc., which negatively affect the techniques' applicability and raises the cost of maintaining the tests [10, 12, 16, 52].

Visual GUI Testing is an emerging technique in industrial practice that combines scripting with image recognition in open source tools such as Sikuli [20], and commercial tools such as JAutomate [67]. Image recognition provides support for user emulated interaction with the bitmap components, e.g. images and buttons, shown to the user on the computer monitor and is therefore perceived to enable testing of any GUI-based system, regardless of implementation, operating system or even platform. Empirical research has shown the technique's industrial applicability for high-level test automation with equal or even better fault finding ability than its manual counterparts [121]. However, VGT is not without its challenges, problems or limitations (CPLs), such

as CPLs related to tool immaturity and image recognition volatility. These CPLs have not been sufficiently explored in the existing studies and it is thus hard to give a balanced description of VGT to industrial practitioners as well as advice and help them in successfully applying it.

In this paper we present an empirical study performed at two different companies within the Swedish corporation Saab, in which two teams transitioned existing, manual system test cases to VGT test scripts. The two projects were independent from each other and in two different sites, one driven by a researcher and the other by industrial practitioners, yet both projects provided corroborating results.

During the study, a total of 58 CPLs were identified that were categorized into 29 groups of mutually exclusive CPLs that affect either the transition to, or usage of, VGT in industrial practice. These groups have implications for the industrial applicability of VGT, e.g. adding frustration and confusion during transition and usage, but also automated testing in general. Consequently, the results of this study add empirical evidence regarding CPLs, which is currently limited, to the general body of knowledge about automated testing [97]. In addition, four general solutions were identified during the study that mitigate or solve roughly half of the identified CPLs related to VGT. Solutions that provide support and guidance to industrial practitioners that intend to evaluate or use the technique. Furthermore, quantitative information was acquired during the study regarding the VGT suites' defect finding ability, the VGT suites' development costs and the projects' evaluated return on investment. Information that provide decision support for industrial practitioners regarding the potential value and cost of transitioning their manual testing to VGT. Based on these results we draw a conclusion regarding the CPLs implications on the cost-effectiveness, value and applicability of VGT for high-level test automation in industry.

Hence, the specific contributions of this work are:

C1: Detailed descriptions of the challenges, problems and limitations (CPLs) that impact either the transition to, or usage of, VGT in industry.

C2: Descriptions of identified practices, from the industrial projects, which solve or mitigate CPLs that impact the transition to, or usage of, VGT in industry.

C3: Quantitative information on the defect finding ability of the developed VGT suites compared to the manual test suites that were used as specification for the automated tests.

C4: Detailed information on the cost of transition, usage and return on investment of the VGT suites, in context of the manual test suite execution cost.

The next section, Section 4.2, presents a background to manual testing and GUI-based testing as well as related work on previously used GUI-based test techniques and VGT. Sections 4.4 presents the results from the study, including identified VGT related CPLs, solutions to said CPLs, VGT bug finding ability, metrics on the cost of transitioning to VGT and ROI metrics. Section 4.5, presents a discussion regarding the results, future work, as well as the threats to validity of the study. Finally Section 7 will conclude the paper.

4.2 Background and Related work

The purpose of high-level testing, e.g. system and acceptance testing, is to verify and/or validate system conformance to a set of measurable objectives for the system, i.e. the system's requirements [25]. These tests are generally performed on a higher-level of system abstraction since their goal is to test the system in its entirety, which also makes them hard to automate, generally requiring large and complex test cases. Tests that are particularly hard to automate are non-functional/quality requirements (NFR) conformance tests for NFRs such as usability, user experience, etc. The reason is because NFRs differ from the functional requirements since they encompass the system in its entirety, i.e. they depend on the properties of a larger subset, or even all, of the functional components. Hence, for a non-functional/quality requirement to be fulfilled, all, or most, of the components of the system have to adhere to that requirement, which is verified either during system or acceptance testing of the system. Both system and acceptance tests are, in general, based around scenarios [37,122], but with the distinction that system tests only aim to verify the functionality of the system. In contrast, acceptance tests aim to validate the system based on end user scenarios, i.e. how the system will be used in its intended domain. Note that we use the word scenario loosely in this context, i.e. not just documented scenarios, e.g. use cases, but ad hoc scenarios as well. These tests should, according to [25, 38, 102, 103], be performed regularly on the system under test (SUT), preferably using automated testing. Automated testing is proposed because both manual system and acceptance testing are suggested to be costly, tedious and error-prone [2–7]. Therefore, a considerable amount of research has been devoted to high-level test automation techniques, which has resulted in both frameworks and tools, including graphical user interface (GUI) based interaction tools [31, 107].

GUI-based test automation has received a lot of academic attention, with research into both high-level functional requirement and NFR conformance testing, as shown by Adamoli et al. [106]. In their work on automated performance testing they identified 50 articles related to automated GUI testing using different techniques. One of the most common they identified was capture/record and replay (R&R) [6,106,107]. R&R is a two step approach, where user input, e.g. mouse and keyboard interaction performed on the SUT, are first captured in a recording, e.g. a script. In the second step, the recording is replayed to automatically interact with the SUT to assert correctness during regression testing. However, different R&R techniques capture recordings on different levels of system abstraction, i.e. from a GUI component level to the actual GUI bitmap level shown to the user on the computer's monitor. The GUI bitmap level R&R techniques drive the test scenarios by replaying interactions at exact coordinates on the monitor, i.e. where the GUI interactions were performed by the user during recording. However, the assertions are generally performed on lower levels of system abstraction, i.e. component level, but some tools also support bitmap comparisons. In contrast, widget/component-based R&R techniques are performed completely on a GUI component level, i.e. by capturing properties of the GUI-components during recording and using these properties, in combination with direct interaction with the GUI components, e.g. invoking clicks, to perform interaction during

playback. However, both of these techniques suffer from different limitations that affect their robustness, usability, cost, but foremost their maintainability, suggested by empirical evidence related to the technique [4,10,13,52,123,124]. The coordinate-based R&R techniques are sensitive to GUI layout change [6], e.g. changing the GUI layout will cause the script to fail, whilst being robust to changes in the code of the tested system. Widget/component-based R&R techniques are instead sensitive to API or code structure change, whilst being robust to GUI layout change [12]. In addition, the technique can pass test cases that a human user would fail, e.g. if a widget blocks another widget, the use of direct component interaction still allows the tool to interact with the hidden widget. Furthermore, because direct component interaction is used, the technique also requires access to the backend of the system and generally only work for SUTs written in one programming language. Some exceptions exist, e.g. TestComplete [125], which support many different programming languages, or even interaction with the Windows operating system, but not other operating systems, e.g. MacOS. Hence, even though previous techniques have properties that support their use for high-level testing, they still suffer from limitations that perceivably limit their use for systems written in different programming languages, distributed systems, and cloud based systems where access to the backend is limited. Thus, indicating that there is a need for more research into high-level test automation.

Visual GUI Testing is an emerging technique in industrial practice that uses tools with image recognition capabilities to interact with the bitmap layer of a system, i.e. what is shown to the user on the computer's monitor. Several VGT tools are available to the market, including both open source, e.g. Sikuli [54], and commercial, e.g. JAutomate [67], but even so the technique is only sparsely used in industry. VGT is performed by either manual development, or recording, of scripts that define the intended interaction with the SUT, usually defined as scenarios, which also include images of the bitmap-components the tools should interact with, e.g. buttons. During script execution, the images are matched, using image recognition, against the SUT's GUI and if a match is found an interaction is performed. This capability is also used to assert system correctness by comparing expected visual output with actual visual output from the SUT. Hence, VGT uses image recognition both to drive the script execution and assert correctness compared to previous techniques where image recognition was only used in some tools for assertions. Furthermore, VGT is a blackbox technique, meaning that it does not require any knowledge of the backend and can therefore interact with any system, regardless of implementation language or development platform, e.g. desktop (Windows, MacOS, Linux, etc.), mobile (iPhone, Android, etc.), web (Javascript, HTML, XML, etc.).

In our previous work [65], we performed an empirical, comparative, study with two VGT tools to identify initial support for the technique's industrial applicability. The tools, Sikuli and CommercialTool[1], were compared based on their static properties as well as their ability to automate industrial test cases for a safety-critical air traffic management system. 10 percent of the tested system's manual test cases were automated with both tools, which showed that there was no statistical significant difference between the tools and that

[1] The name of CommercialTool can not be disclosed due to confidentiality reasons.

both tools were fully capable of performing the automation with equal fault finding ability as the manual test cases. However, the study was performed by academic experts in VGT and therefore a second study was performed, driven by industrial practitioners in an industrial project [121]. Results of the second study showed that VGT is also applicable when performed in a real project environment when used by industrial practitioners under real-world time and cost constraints. Thus, providing further support that VGT is applicable in industry.

A general conception within the software engineering community is that automated testing is key to solving all of industry's test-related problems. However, Rafi et al. [97] that performed a systematic literature review regarding the benefits and limitations of automated testing, as well as an industrial survey on the subject, found little support for this claim. In their work, they scanned 24.706 academic reports but only found 25 reports with empirical evidence of the benefits and limitations of automated testing. Additionally, they found that most empirical work focus on benefits of automation rather than the limitations. Furthermore, the survey they conducted showed that 80 percent of the industrial participants, 115 participants in total, were opposed to fully automating all testing. In addition, they found that the industrial practitioners experienced a lack of tool support for automated testing, e.g. the tools are not applicable in their context, have high learning curves, etc. Consequently, their work shows that there are gaps in the academic body of knowledge regarding empirical work focusing on the benefits and in particular the limitations of automated testing as well as the actual needs for test automation in industry. A gap that this work helps to bridge by explicitly reporting on the challenges, problems and limitations related to high-level test automation using VGT.

4.3 Industrial case study

The empirical study presented in this report consisted of two parts. First, a VGT transition project driven by a researcher at the company Saab AB, in the swedish city of Gothenburg, building on our previous work [65]. Second, a VGT transition project in another Saab company within the Saab corporation, in the swedish city of Järfälla, which was driven by industrial practitioners with minimal support by the research team. Hence, the two projects complement each other by providing information of academic rigor from the project driven by the researcher, and information from the practical usage of VGT from the second project driven by industrial practitioners. Thus, this paper presents the results from two holistic case studies [17] from two different companies, in the continuation of this report referred to as Case 1, the study in Gothenburg, and Case 2, the study in Järfälla. The two case studies were conducted in parallel, but with different research units of analysis [17]. In Case 1 the unit of analysis was the VGT transition of one complete manual system test suite of a safety-critical air traffic management system, performed by experts from academia. In Case 2, the unit of analysis was instead the success or failure of the VGT transition project when performed in a practical context by industrial practitioners under real-world time and cost constraints. Consequently, Case

1 provided more detailed information about the VGT transition, whilst Case 2 provided information from a VGT transition project performed by industrial practitioners under practical conditions.

4.3.1 The industrial projects

Both VGT transition projects were conducted in industry with two mature industrial software products/systems. In Case 1, the VGT transition was performed on a distributed, safety-critical, air traffic management system with an excess of 1 million lines of code, highly configurable to satisfy customer needs, developed in a multiple programming languages. Additionally, the system was graphical user interface (GUI) driven with a very shallow GUI, meaning that most graphical bitmap components were continuously shown to the user, i.e. the GUI did not change much during interaction. The system has been developed using both plan-driven and iterative development processes, starting with requirements acquisition activities, followed by development activities and finally testing activities. For each activity a set of artifacts were developed, such as requirements specifications, design documents, user guides, test documentation, etc. However, for the study, the only document of interest was the manual system test descriptions, which specifies the manual test cases used for system testing. System testing that is generally performed once or twice every development iteration, i.e. once or twice every six months. Furthermore, the company's developers are highly-educated and most have many years of industrial experience as different roles, e.g. as developers, testers and project managers. However, the company does not use dedicated testers, instead the developers perform the testing. The company has a hierarchal, yet flexible, organization, distributed between two locations, i.e. Gothenburg and Växsjö. However, Case 1, due to resource constraints was only performed in Gothenburg. Furthermore, the system used in the project was developed according to a quality assurance process that is compliant with the RTCA DO-278 quality standard. A standard required by many of the system's customers, which consist of both domestic, public and military, airports as well as international, public, airports.

In Case 2 the VGT transition was performed on manual test cases for a battlefield control system which is distributed over several computers and is both safety- and mission-critical. However, due to the limited involvement of the research team in Case 2, and confidentiality reasons, less can be disclosed about the system's details. The product is however mature, i.e. it has been developed, maintained and deployed for many years. Additionally, the tested system is GUI-driven but with a deeper GUI meaning that the entire view of the GUI can change during interaction with the system, e.g. when opening menus or new windows. Development of the system was, at the time of the study, performed using an iterative process with regular manual testing performed by dedicated testers. However, in contrast to the system developed in Case 1, the tested system in Case 2 did not follow any quality assurance standard. The look and feel of the graphical bitmap components of the system's GUI were however specified by a military standard. Finally, the product's main customer is a Swedish military contractor.

Figure 4.1 visualizes the research process and the three parallel tracks that

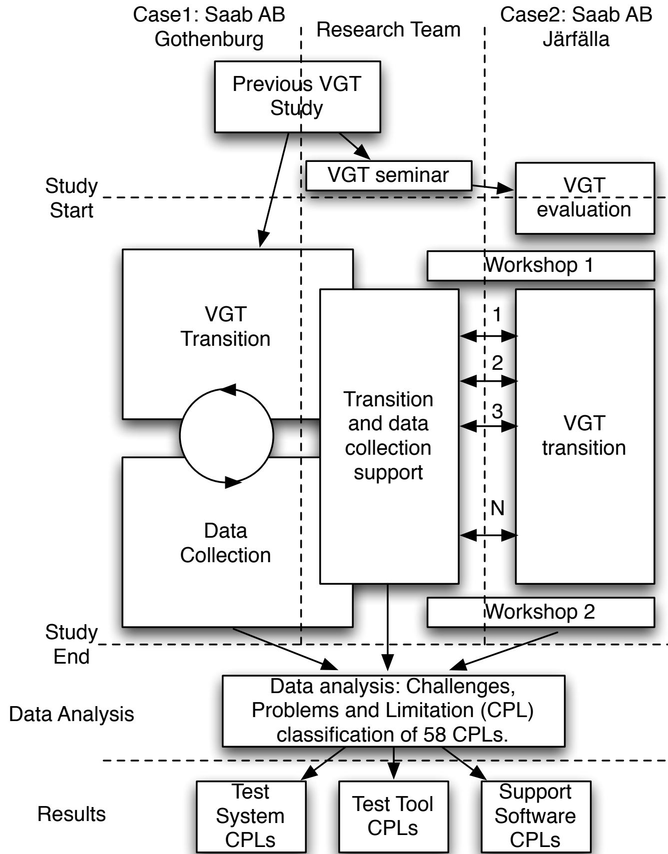

Figure 4.1: *Visualization of the research process showing three tracks. Track 1 contains activities performed in Case 1 and prior work at Saab AB in Gothenburg. Track 2 the activities of the research team and Track 3 the activities of Saab AB in Järfälla. Boxes that cross over the dotted lines were performed by the research team and the respective company.*

were performed by the researchers in Case 1, practitioners in Case 2 and the support and data acquisition performed by the research team. The support activities were continuous in Case 1, with members of the research team on site, at the company, daily during the project. In Case 2, personal support was only given during two full-day workshops on site, whilst all other support activities were conducted over email or telephone. The workshops performed in Case 2 were also the main source of data acquisition from the company, e.g. information such as what type of system they were working with, what the Sikuli test architecture would look like, etc. In the second workshop, two semi-structured, one hour, interviews were held with the industrial practitioners to validate previously acquired information through triangulation. Consequently, the data acquisition in Case 2 was divided into three distinct phases, introduction (Workshop 1, in phase 1), data acquisition and implementation support through remote communication (Phase 2) and finally a retrospective analysis

(Workshop 2, phase 3).

After the completion of Case 1 and Case 2, all collected/acquired information was analyzed and the CPLs identified. In Case 1, the analysis was done through a combination of discussions with the researcher who did the data collection and analysis of the thorough documentation that had been kept during the entire project. In Case 2, document analysis was used as well, but the primary source of information came from the workshops and particularly the interviews since explicit questions were asked regarding the CPLs, and solutions to said CPLs. The two projects were performed mutually exclusive from one another, meaning that the leading researcher in Case 1 had no interaction with the industrial practitioners in Case 2. Thereby ensuring that any corroborating evidence from the two projects were not influenced by each other.

After identification, the CPLs were categorized into three tiers, with 29 mutually exclusive CPLs on the lowest level of abstraction, i.e. Tier 3. The Tier 3 CPLs were then generalized into 8 groups, Tier 2 CPLs, which could be grouped even further into three top tier CPLs, i.e. Tier 1 CPLs. Furthermore, the Tier 3 CPLs were analyzed to identify which were the most prominent CPLs in the projects. Prominence was evaluated based on occurrence in both projects, as well as more subjective measures, e.g. perceived negative impact on the transition, or usage, of VGT, added frustration to the VGT transition, and perceived external validity. In addition this analysis revealed four high-level generic solutions that had been identified during the two projects, which solve or mitigate roughly 50 percent of the identified CPLs. Cost, and return on investment, was also evaluated during the study based on the quantitative metrics that were collected from both cases and then analyzed. This analysis was performed in context of the acquired qualitative information, e.g. the industrial practitioner's statements regarding VGT's benefits and drawbacks, to rule out bias by ensuring that the qualitative and quantitative information was coherent. All information analysis was performed by another member of the research team than the one who performed the transition project in Case 1 to mitigate bias in the results. Finally, conclusions were drawn from the analyzed information, which were reviewed and validated by the researcher who did the data collection and industrial practitioners to eliminate bias introduced during the analysis.

4.3.2 Detailed data collection in Case 1

The VGT transition in Case 1 started with an analysis of the automated test scripts that were developed in our previous work at Saab in Gothenburg to identify what should/could be reused in the new project. In parallel with this analysis, a thorough analysis was performed of the manual test suite for the tested system. The document analysis was necessary because the version of the tested system differed from the system that had previously been used. In addition, the document analysis was required because the researcher who did the data collection was unfamiliar with the system and lacked the details of the previous work. A thorough documentation process was put in place at this stage of the project based around a set quantitative and qualitative metrics that were collected for each developed VGT script and/or the VGT test

suite, including development time, execution times, CPLs, etc. Specifically, the information collection focused on sources and causes of CPLs that affected the VGT transition, e.g. was the CPL related to the script, the VGT tool or the system.

Even though the version of the tested system in Case 1 differed from the one used in our previous work, the core functionality of the new version was the same as the old, i.e. airport landing and air traffic management. The main difference was that the new version only had one control position, whilst the old system had three, each with different capabilities for different operators. Consequently, the new version had limited functionality and analysis of the manual tests therefore showed that only 33 out of the 50 manual test cases, mentioned in previous work, were applicable. However, all of these 33, applicable, test cases could be automated and thereby constituted a full automated test suite, VGT suite, for that particular version of the system.

Furthermore, the system was distributed over three physical computers, which required the VGT tool, Sikuli, to be paired with a third party Virtual Network Connection (VNC) application in order to perform the test cases. Thus, the test system setup included three computers that were interconnected, through a local area network (LAN), to a fourth computer that ran a VNC viewer application and Sikuli. The test system setup has been visualized in Figure 4.2. Consequently, Sikuli, rather than to execute scripts locally, executed the scripts through the VNC viewer application to facilitate test script execution of distributed test cases that required interaction with more than one computer.

Once the analysis of the manual test suite, and the setup of the test system had been completed, an architecture was defined for the development of the VGT suite. The VGT suite architecture consisted of a main script that imported the individual test case scripts, and helper scripts, and executed these sequentially. The VGT scripts were implemented one at a time using the manual test cases as a specification, ensuring that each VGT script was a 1-to-1 mapped representation of the manual test case. This implementation choice was possible because the test steps in the manual test cases were defined in sequential scenarios. Hence, each test step of an automated test script became directly traceable to an equivalent manual test step in a manual test case. Additionally, to verify the automated test cases, the test scripts were executed after each test step had been developed, and verified against the results of corresponding manual test step. Furthermore, the automated test steps were defined in mutually exclusive methods, written in Sikuli script which is a scripting language based on Python. This approach made the scripts modular, allowing test steps to be reused. Modularity was one of the keywords for the transition projects to ensure reusability and maintainability of the scripts. Finally, after a test script had completed, and successfully executed against the tested system, it was integrated into the VGT suite. In situations where erroneous test script behavior was identified post-integration into the VGT suite, the test scripts were corrected and validated during execution of the entire VGT suite, i.e. execution of all the test cases in sequence. The validation was rigorously performed, even though time consuming, to ensure high quality, by comparing the outcome of each automated test step with the outcome of corresponding manual test step.

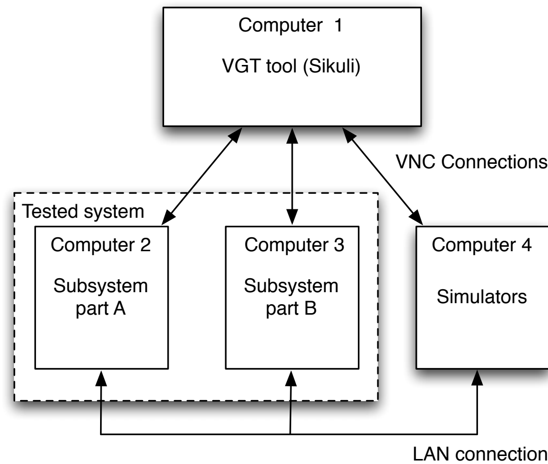

Figure 4.2: *The setup of the tested system, including all computers that were connected in a local area network (LAN), accessed by Sikuli using VNC.*

A second keyword for the VGT transition project in Case 1 was robustness. Robustness was achieved by implementing the scripts with three levels of failure redundancy for critical functions to mitigate catastrophic failure either due to image recognition failure, script failure, test system failure or detection of a defect in the tested system. In addition, all scripts were written as modular as possible to ensure reusability of generic functions. Modularization also made it possible to run specific test steps out of order, thereby shortening script verification, since the entire test script did not have to be re-executed every time new functionality had been added or changed.

In addition, to increase the fault finding ability of the VGT suite, a third party screen-capture software was integrated into it. The screen-capture software was used when a script failed, which would cause the tested system to automatically reset to a known state after which the test scenario would be rerun whilst being recorded by the screen-capture software. The recording functionality simplified script verification and was also used to identify defects in the tested system, i.e. if a script failed, a new video clip was recorded and saved which the developers could view in order to recreate the defect. In addition to the he video-recordings, textual log files were created and saved for each executed test case.

4.3.3 Detailed data collection in Case 2

In contrast to Case 1, Case 2 was driven by industrial practitioners and started with a three week long evaluation of VGT as a technique. Three tools were evaluated during this period, i.e. Sikuli, eggPlant and Squish, to identify the most suitable tool to fulfill the company's needs. After the evaluation, and because of recommendations from the research team, Sikuli was identified as the most suitable alternative for the automation.

Similarly to the transition in Case 1, the industrial practitioners in Case 2 used their manual test cases as specifications for the automated test cases.

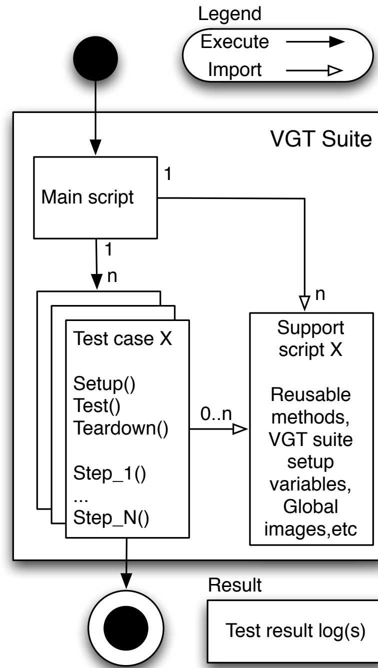

Figure 4.3: *The architecture of the VGT suites developed in Case 1 and Case 2.*

However, since the testers possessed expert domain knowledge, not every test case was implemented as a 1-to-1 mapping to the equivalent, manual, test case. The deviations from the 1-to-1 mappings were required since the manual tests in Case 2, defined as use cases, were not mutually exclusive, but rather linked together into test chains that contained several test flows, where each test flow constituted a test case. The manual test case architecture in Case 2 was however perceived to provide the VGT suite with a higher degree of flexibility and reusability than the manual test architecture used in Case 1. However, similar to the VGT suite developed in Case 1, the developed VGT suite in Case 2 was based around a main script that imported individual test cases and executed these according to a predefined order that was specified by the user.

Furthermore, similar to Case 1, quantitative metrics and qualitative information was collected during the project. However, in contrast to the systematic, and rigorous, information collection in Case 1, it was performed in an ad hoc fashion in Case 2 because of time and cost constrains. The collected information was then conveyed to the research team, as shown in Figure 4.1, through e-mail, telephone or during the interviews in Phase 3.

4.3.4 The VGT suite

In this this study we do not consider the developed VGT suites, or their architecture, part of the contributions of the work since the main focus of the

study was on the challenges, problems and limitations (CPLs) that were identified during their development and usage. However, to provide background, and replicability, of the study, the following section will describe the developed VGT suites in more detail.

The VGT suites developed in Case 1 and Case 2 were similar in terms of architecture and were both built in Sikuli script, which is based on Python. Sikuli has support for writing individual VGT based unit tests and includes special assertion methods for unit testing. However, Sikuli does not have support for creating test suites of several individual unit tests. Hence, in order to create a VGT test suite of unit tests, using Sikuli's supported functionality, all the tests have to be grouped into one large script, which has negative effects on reusability, maintainability, usability, etc, for large suites. Therefore, custom test suite solutions were created in both projects by using Python's support for object orientation and its ability to import scripts into other scripts. Therefore, both VGT suites consisted of a main script that imported the individual test cases and executed these according to an order specified manually in a list in the main script. The architecture for the VGT suites is visualized in Figure 4.3. As can be seen in the figure, each script was given a setup method, a test method, containing the test steps of each test case, and a teardown method. Consequently, the VGT scripts were developed to follow the same guidelines used for automated unit tests, e.g. JUnit [11]. Additionally, user defined methods, variables to setup the VGT suite, etc, were extracted from the individual scripts and put in a set of support scripts that were then imported to the main script and/or the test scripts that required them. The modular, and hierarchal, architecture helped shorten development time, increase reusability and improve maintainability of the scripts, since all reusable components were grouped in one location.

Test assertions were conducted through visual comparison between expected and actual output from the SUT using Sikuli's image recognition algorithm. This was achieved using branch statements, i.e. if the expected output was observed the test step passed, else it failed.

The key difference between the VGT suite developed in Case 1 compared to Case 2 was how test result output was generated. In Case 1, the output was generated as textual log files using a custom solution that was spread across the main script and the individual test scripts. The solution documented the results of individual test steps but also summarized the results from the entire test case, i.e. providing feedback to the developers on two levels of abstraction. In addition, the output included video recordings of failed script executions, created using the third party recording software Camtasia.

In Case 2, the output was produced using an open source Python library that formatted the output from the test scripts into an HTML format, i.e. providing a graphical representation of the test results, similar to the output from automated unit tests, e.g. JUnit [11]. However, in contrast to Case 1, Case 2's VGT suite did not record failed test scenarios, instead it only took screenshots of the tested system's faulty state when a test case failed, i.e. capturing the GUI's faulty state when a fault occurred.

The purpose of adding screenshots, and video recordings, to the VGT suites' output was to provide the developers of the tested system with more information to simplify fault identification and recreation. This functionality

also helped the testers to distinguish defects in the tested system from faults in the VGT suite itself, either caused by faulty test case implementation or image recognition failure during test script execution. Hence, mitigating the risk of false positives, i.e. reporting defects that were actually not defects.

4.4 Results and Analysis

During the study, 58 challenges, problems and limitations (CPLs) were identified related to the transition, or usage, of VGT. These CPLs were categorized post-project completion into three tiers of VGT related CPLs, as shown in Figure 4.4. Three main categories of CPLs were identified, i.e. CPLs related to the tested system, the VGT tool or support software, constituting the Tier 1 CPLs. The Tier 1 CPLs were then split into eight CPL sub categories based on their origin and/or root of cause, i.e. the Tier 2 CPLs. These eight Tier 2 CPLs are: CPLs related to the version of the test system, the general test system, defects in the test system, CPLs related to the practices of the company, the tested system's simulators, Sikuli, the test scripts or third party software. In Figure 4.4 these eight categories have been divided even further into a third level of abstraction, consisting of 29 identified mutually exclusive groups of CPLs, i.e. Tier 3 CPLs.

Out of the 58 identified CPLs, 14 were identified as Sikuli specific. Five of the 14 Sikuli CPLs were unique, i.e. mutually exclusive from any other Sikuli CPL, eight were related to image recognition failure or tool volatility and the last CPL related to the developed VGT test scripts. Furthermore, 20 out of the 58 identified CPLs were related to the version of the tested system, six related to the general system, i.e. the product, six were defects in the tested version of the system, one to the company in general and one to the simulator environment, i.e. 34 in total. Hence, more than twice as many of the identified CPLs originated from the tested system compared to the tool, i.e. Sikuli. The remaining 10 CPLs were related to the third party software that was used in order to realize the VGT suite, e.g. the virtual network connection (VNC) application used to implement distributed test cases and the recording software, Camtasia.

The detailed information regarding the CPLs were identified primarily in Case 1, but were corroborated by information acquired in Case 2. Hence, as can be seen in Figure 4.4, only 18 out of the 29 mutually exclusive CPLs were identified in both cases, i.e. 62 percent of the CPLs. However, the two cases were performed mutually exclusively of one another, meaning that the researcher who did the data collection in Case 1 had no contact with the industrial practitioners in Case 2. This study design intended to ensure that the collected information was based on the individual cases and could corroborate each other. Hence, provide evidence to support that the identified CPLs are generic for any VGT project performed with Sikuli.

In the following sections details regarding the identified CPLs will be presented. The presentation will use Case 1 as a base case but corroborating information from Case 2 will be presented as well.

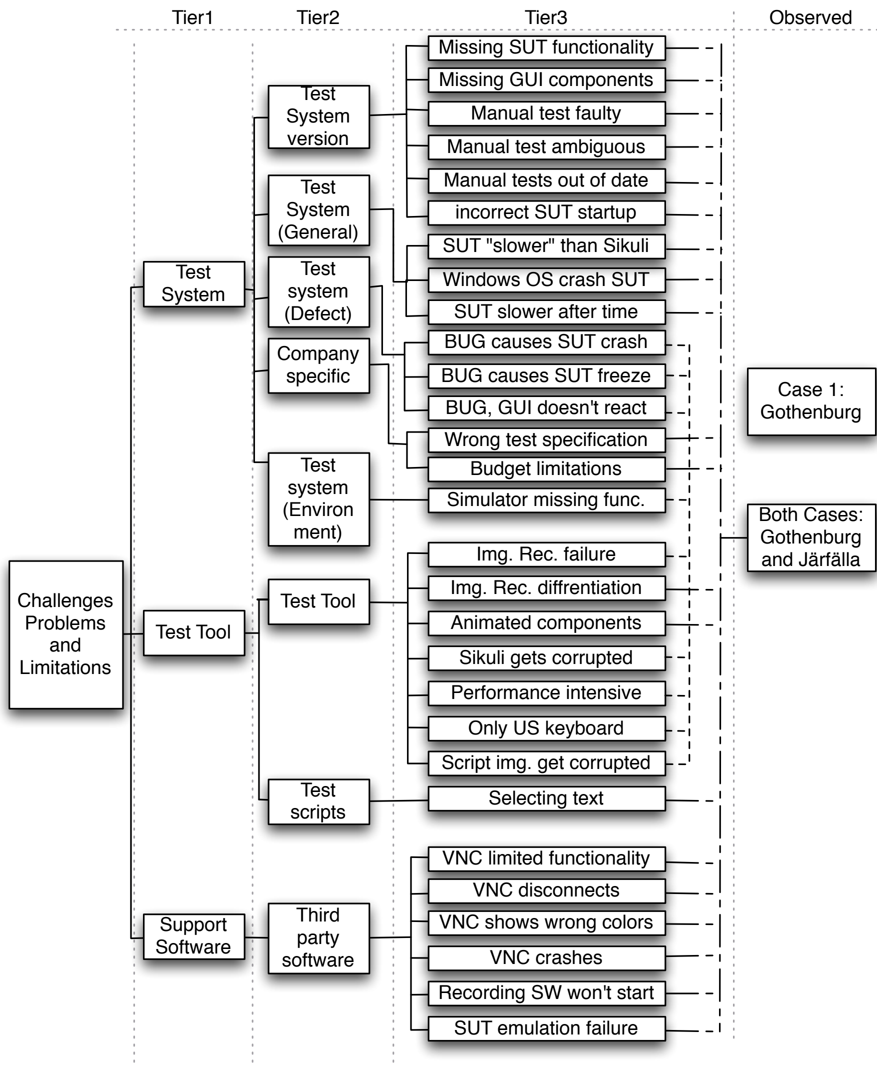

Figure 4.4: *A hierarchal tree diagram over the Challenges, Problems and Limitations (CPLs) that were identified in Case 1 and Case 2. The CPLs have been divided into three tiers of abstraction, with Tier 3 being the lowest level where the 58 identified CPLs have been grouped into 29 groups of mutually exclusive groups of CPLs. The model continues to the right (The grayed out symbols), connecting to Figure 4.5, which shows potential solutions to the Tier 3 CPLs.* ***SUT*** *- System under test,* ***VNC*** *- Virtual Network Connection,* ***Img. Rec.*** *- Image recognition,* ***GUI*** *- Graphical User Interface,* ***OS*** *- Operating system,* ***SW*** *- Software,* ***Func.*** *- Functionality.*

4.4.1 Test system related CPLs

The identified test system related CPLs were split into five sub-categories, as shown in Figure 4.4. These CPLs are related to the version of the test system,

the general test system, defects in the test system, specific to the company or the test system's simulators. In this section, a summary of each CPL subcategory has been described, including data related to the individual CPLs of each category.

4.4.1.1 Test system version

Testing of complex systems with several versions and/or variants is related to several CPLs that also affect VGT. One such CPL relates to the manual test specification, e.g. that it can be faulty, out of date or developed for another version or variant of the tested system, i.e. not aligned with the tested system. This CPL appeared in Case 1 and was caused by the build, i.e. version, of system used in the project which was a demo system used to demonstrate the functionality of the product to potential customers, i.e. not a system intended for customer delivery. As a consequence, the demo system was limited in terms of functionality, lacked documentation, etc. The reason why this system was used in Case 1 was because of resource constraints. Whilst the product of the system required 12 computers, including three redundant servers, the demo system required only three computers in total. Furthermore, because the VGT suite was implemented as a 1-to-1 mapping of the manual test suite, not all of the system's manual test cases could be automated. In our previous work [126] we performed automation with VGT on a version of the tested system that had 50 applicable manual test cases. However, for the product of the tested system, i.e. the full system, there are 67 manual system test cases in total of which only 33 could be automated in Case 1. Hence, restricting the usability and portability of the developed VGT test suite for other versions or variants of the product system. However, all of the 33 test cases were automated and thereby constituted a complete automated system test suite for demo system.

Lack of functionality that prohibited automation of more manual test cases were, but not limited to, the system version's lack of operator roles, missing GUI components, missing radar functionality, missing simulator support, etc. This functionality had purposely been omitted by the company when the demo system was developed to scale down the amount of required hardware.

Furthermore, several CPLs were related to the manual test cases for the tested system, e.g. some test cases were incorrect or out of date. Other tests were found to be ambiguous or aimed at testing functionality that was no longer part of any variant or version of the system. These test cases were reported to the company and were considered a positive side effect of the automation since the faulty tests could be removed from the test specifications for several versions or variants of the system. Thus, this CPL relates to the complexity of keeping test specifications up to date as a mature system evolves into versions/variants over time.

Another identified CPL originates from the developers'/testers' lack of domain knowledge in combination with ambiguity in the manual test descriptions/specifications. If a developer/tester performing the VGT transition lacks domain knowledge, it may be impossible for him/her to implement certain tests or he/she might implement them incorrectly due to ambiguities in the test specification, whilst a domain expert would have been able to resolve the CPL. This CPL was observed in Case 1, where the test specification that was

used for the test automation was intended for another version of the tested system, making several of the test cases inapplicable. Hence, many of the test cases could only partially be implemented, or not be implemented at all, because the tested system lacked functionality. Furthermore, this made the VGT script implementation time consuming since the researcher who did the data collection continuously had to ask other developers at the company what the ambiguous test cases referred to, or what went wrong when the test cases did not align with the system specification. To resolve the CPL the test specification that was initially used in Case 1 was replaced with another version that was perceived to be better aligned with the system. However, it was soon found that the new version of the test specification included other test cases that could not be implemented on the version of the tested system used in the project, i.e. causing new CPLs. The new CPLs were not discovered before the automation had already started, once again due to the researcher who did the data collection's lack of domain knowledge. If the VGT transition had instead been performed by a domain expert, he or she might have been able to identify earlier that the test case was not applicable. Thus, saving time and cost. This assumption is supported by information from Case 2 where the industrial practitioner's reported no such problems since they had extended knowledge of both the test cases and the SUT.

Several other CPLs were identified during analysis of the manual test specification, even though most of the CPLs were uncovered during script development. These CPLs were related to defects in the tested system that caused the VGT scripts to terminate with a certain probability or every time they were executed. During real-world execution of the VGT suite, i.e. during regression testing, a VGT script would terminate after identifying a defect, report the defect in the test output log, followed by a roll-back of the system to a known state before executing the next script. Consequently, a manual test case that identifies a defect at test step n can in practice only be automated up to step n since the defect would prohibit any interaction with the tested system after this step. This is because all further execution, after identifying a defect, would be within an unknown, and potentially useless, system state, making the interaction at test step n+1 invalid and/or useless compared to real-world use of the system. Furthermore, continued execution has a high probability of reporting false positive results because the test steps are generally dependent on one another, i.e. test step x sets the system in a state that is asserted in test step x+1 that also sets the system in a state that is required in test step x+2, etc. However, since the VGT suite, in Case 1, was developed for future use at Saab, all test steps had to be implemented regardless if they succeeded a faulty test step or not. Thereby, raising the usability and portability of the developed VGT suite for versions of the tested system where the fault had been corrected. To ensure that the test steps after a defect had been identified would still be performed during script execution, the assertion of the defect finding step, e.g. test step n, was disabled. Thus, test step n could perform the necessary interaction with the SUT required for test step n+1. However, since these test cases ignored the defects, the continued script execution, i.e. test steps n+1 and forward, potentially put the system in an invalid/unnatural state. Six defects were found during Case 1, by six different manual test cases, resulting in six partially implemented test scripts, i.e. scripts with disabled

CPL category	**CPL sub-category**	**Nr. of occurrences**	**Percentage of all CPLs**
Test System	Test System version	20	0.344
Test System	Test System (General)	6	0.103
Test System	Test System (Defects)	6	0.103
Test System	Test Company specific	1	0.0172
Test System	Test System (Environment)	1	0.0172
	Total	37	0.5844

Table 4.1: *Summary, and distribution, of problems, challenges and limitations (CPLs) related to the tested system.*

assertions. Hence, in future use of the test suite, for other versions or variants of the system, these assertions first need to be enabled.

The defect identification was continuous in four of the six developed test scripts that found defects. To ensure that these test scripts would execute all their test steps, the assertions that found the defects were disabled during verification of the scripts. This was possible because the test steps were mutually exclusive from one another and because of the shallow GUI design of the tested system that continuously showed all the GUI's bitmap components to the user. However, in the remaining two test cases the tested system's faulty behavior was not consistent. In these test cases the assertions were left active during verification of script functionality which gave the test scripts a certain probability of failure. This also made it possible to identify the faults in sufficient detail to report them to the company.

However, because of CPLs related to faulty system behavior, it is reasonable to assume that VGT transition of systems early in their development cycle, i.e. VGT transition of immature systems, lead to more partially implemented test scripts because some assertions cannot be implemented. The reason is because a VGT script, like previous GUI-based techniques and manual system tests, aim to test the system in its entirety, and therefore requires the system to have reached a reasonable level of maturity in order to be applicable [127].

It is also perceived that an incremental script development process in parallel with system development has benefits over a big bang script implementation. This assumption is based on the research into other automated test techniques and practices [16, 33], in particular Test-driven Development that suggests continuous, and incremental, test script development [128]. However, as shown in this study, a big bang implementation for legacy systems is also considered viable. We will return to the viability of this approach in Section 4.4.6, which discusses the return on investment of the VGT transition in Case 1 and Case 2. However, even though these assumptions are supported by research in other areas of automated testing, future work is still required to verify their affects on VGT.

4.4.1.2 Test system (General)

One of the most time consuming CPLs that was identified, in both Case 1 and Case 2, was that Sikuli, due to the speed of Sikuli's image recognition algorithm, executes scripts faster than the system's GUI can respond. Consequently, when Sikuli tries to perform an interaction, it is not certain that the GUI is ready to receive new input, which can cause the script to fail. This CPL is time-consuming because it requires the user to add delays in the scripts to synchronize the scripts' execution with the tested system. Furthermore, this practice slows down the script execution time since the delays either have to wait for the GUI to reach a stable state or be based on fixed delays based on the worst case responce time of the system. Furthermore, this practice, even though it adds robustness, usability, reusability and portability to the VGT suite it also adds maintenance costs since these delays generally have to be maintained every time the system's performance is changed. In addition, as reported by the practitioners in Case 2, this practice adds frustration to the VGT transition, especially for long test scenarios, since each synchronization, i.e. added delay, requires the script to be rerun to verify correctness. Consequently, If an added delay is not sufficient to synchronize the script with the system's execution, the script once again has to be rerun. In addition, since the delays should preferably be as short as possible, but still ensure script robustness, this practice requires some trial and error which is both tedious and time consuming.

To mitigate this problem, most VGT tools, including Sikuli, have special methods that delay the script execution until a sought image is found, i.e. the script is delayed until a stable GUI state has been reached. However, these methods still require fine-tuning and therefore do not completely remove the synchronization CPL. For web-based systems, this CPL is especially problematic since network latency has to be taken into consideration when adding the delays, i.e. sudden dips in network latency, if not mitigated in the script, can cause the scripts to fail. Hence, a conclusion can be drawn that the execution speed of a VGT script is governed by the response time of the tested system. This conclusion is also corroborated by information collected in Case 1.

This problem became even more troublesome in Case 1 because it was observed that the tested system lost performance over time, i.e. its response rate deteriorated after a couple of days if the computers were not restarted. Similar reports were presented from Case 2 and had the effect that test scripts that had previously succeeded stopped working. However, given that the solution was to simply restart the computer(s) this CPL is considered minor. However, since only minor dips in response rate can cause the scripts to fail, which might be hard for a human to detect, this CPL is worth mentioning since it caused frustration during the project.

In addition, as reported by the practitioners in Case 2, a common comment by project managers and other developers viewing the automated script execution was: "Isn't the execution faster than this?". This comment reflects, and reveals, an important perception about automated test techniques, i.e. the perception that all automated tests execute very fast. This perception is mostly true, e.g. automated unit tests can execute many hundreds, or even of thousands, of test case lines of code in minutes [33]. However, these auto-

mated test techniques, e.g. unit testing, approach testing from a lower level of system abstraction, e.g. through a white-box approach that stimulate the components directly, but out of context. In addition, these techniques do not provide interaction stimuli to the tested system in the same way as the end-user of the system. For instance, a unit test generally only tests one or a couple of components at once, thereby disregarding the timing constraints between the components that appear during actual usage of the system. Attempts have been made to use lower level test techniques, e.g. unit tests, for system test automation. However, these tests have quickly become very large and complex and therefore unmaintainable [15, 16]. VGT scripts take these timing constraints into consideration by emulating end user interaction, but, as mentioned, at the cost of test execution speed. However, based on information from both Case 1 and Case 2, VGT tests can still execute up to 16 times faster than their manual equivalent tests. Hence, VGT scripts execute considerably slower than unit tests, which may be a CPL for industrial VGT adoption since the general perception of industrial practitioners, as mentioned, is that automated testing is vert fast. However, what must also be considered is the higher level of system abstraction of these tests, which make them applicable for system and acceptance test automation.

Another CPL, which is general for all systems, relates to unexpected system behavior, which includes the behavior of the tested system itself, behavior of the operating system or third party software used to run the tested system or its surrounding environment. In Case 1, such CPLs appeared at several occasions but especially in test cases that required reconfiguration of the tested system. The reason was because reconfiguration required both the tested system and the computers' operating system (OS) to be restarted. When the OS was shut down, one or several of the tested system's services often crashed, causing the operating system to launch a pop-up message asking the user to terminate the process. The pop-ups blocked, or delayed, other components in the GUI from becoming visible and were thereby not available for the image recognition algorithm to find, causing the scripts to fail. Events, similar to this scenario can be handled using exception handling but it is often difficult, or even impossible, to anticipate when this type of unexpected behavior will occur, especially if the unexpected behavior is sparse. However, failure to mitigate these events can cause the entire VGT suite to fail, or report false positive faults. Scenarios when this type of behavior occur are when popups, e.g. software update messages or error messages are launched by the operating system, when system performance suddenly drops due to, for instance, a new process is started by the operating system, hardware failure or a defect in the tested system. To mitigate catastrophic script failure due to unexpected behavior, or because of identified defects, the VGT suite in Case 1 was implemented with triple failure mitigation redundancy. First, certain operations, e.g. delay operations based on image recognition, were encapsulated by exception handling blocks that, if triggered, tried to redo the operation on an individual test script level. Second, if the exception handling failed, the exception would be sent up one level in the VGT suite architecture to the main script that would try to roll back the system to a known state and then rerun the test script from the beginning. Third, if the rollback failed, the main script would restart the tested system to ensure a stable system state and

then rerun the failed test case from the beginning. This solution added extra execution time to the overall test suite, but also added confidence that found defects were actually defects in the system rather than spontaneous image recognition failures or faults in the test scripts. Hence, if all three levels of failure mitigation failed to resume the test execution, the cause was in general ruled out to be a defect in the tested system. Similar practices were used in Case 2, but only with one level of failure mitigation.

4.4.1.3 Test system (Defects)

Many of the encountered defects, discussed in Section 4.4.1.1, were previously known to the company in Case 1. However, during the project, defects were also uncovered that were previously unknown or only partially known to the company. Hence, defects that had not been corrected in later versions or variants of the tested system. The reason why these previously unknown defects were uncovered by VGT was because of the quicker and more cost effective execution of the VGT suite compared to manual tests. One reason was because some of the identified faults were not consistent, i.e. could not always be replicated, neither manually or automatically. For instance, one such defect regarded a tab menu in the tested system that did not always load properly. However, since there is no cost related to running the VGT suite it could be run several times in a row and thereby force the faulty system behavior to appear. In combination with the video recording functionality of the VGT suite in Case 1, the faulty behavior of the system could be determined and the defects could be reported in more detail to the company. Hence, showing how the video recording functionality of the VGT suite adds to its fault-finding ability and thereby usability. The faults that were uncovered were of different nature, from faulty GUI functionality, e.g. tabs not launching, to complex faults in the services of the tested system's backend, e.g. missing alarms intended to warn the system user of incorrect or missing input from external interfaces. However, even though some of the defects could be considered minor, the tested system was safety-critical and therefore all faults or defects were considered critical.

Similar information regarding the identification of system defects was acquired from Case 2, where the speed and low execution cost of the VGT suite made it possible to identify three previously unknown defects in the tested system. However, in Case 2 the cause to why the defects had previously not been identified differed from the cause in Case 1. Previous to the automation in Case 2 the test cases were always executed in a linear order, i.e. starting with test case 1 to test case n, and only executed once every development iteration, which was generally six months to one year. Furthermore, due to strict testing budgets, it was not feasible to run all of the system's manual test cases each development iteration. However, the automated tests, which were equivalent to these manual tests, could be executed several times every week and several times in row each time, which lead to the discovery of the previously unknown defects. Therefore, the practitioners in Case 2 reported that if it hadn't been for VGT these defects had probably never been found.

Even though these examples show strengths with the technique, they also present a possible CPL, i.e. the CPL that a company may become so reliant

on the technique that it is used as a substitute for manual testing. However, VGT is not perceived to be able to replace manual testing, primarily because the technique can only find faults that are specified in the test scenarios [16]. Hence, manual exploratory testing is still required in order to complement VGT to uncover new defects in the tested system [42]. These statements were also supported by the industrial practitioners in Case 2.

Furthermore, the information regarding defect and fault-finding indicates that the order of test case execution can affect the fault-finding ability of the scripts, e.g. executing test case b after a may may not have the same outcome as executing a after b. Thus, providing support for our previous statement, i.e. since there is no cost associated with running a VGT suite it can help uncover new defects by supporting execution several times in a row with different predetermined, or even random, test case orderings. Thus, perceivably, increase their effectiveness [9] by covering more system states and sequences of interaction that may appear during real-world usage of the system. However, some of these sequences may be invalid, i.e. never appearing in practical use, but potentially still add value to the development company by exposing what features of the system, in what scenarios, cause faulty system behavior. Consequently, the capability of VGT suites to be executed several times in different sequences, without additional cost, add to their usability. However, in order for the suites to do so effectively they require a good architecture. Thus presenting VGT suite developers with a CPL that is related to traditional software development rather than testing. A CPL that is common to most script-based test automation that also presents new CPLs, for instance how to design a test suite in a good and modular way and how to verify that the test code is correct. In particular, verification of test cases is a difficult CPL since the scripts need to be trustworthy, i.e. not find false positive and absolutely not false negatives. One way of performing verification of VGT tests is to verify them against the manual test descriptions, i.e. ensure that the VGT scripts perform in the same way as their manual test equivalents.

4.4.1.4 Test company specific CPLs

There was a mismatch between the tested system in Case 1 and the test specification that was used for the VGT transition. The researcher driving the automation in Case 1 also lacked the domain knowledge to identify this mismatch which later required several partial test cases to be refactored. However, the source of this CPL is not VGT specific but rather a general CPL that is related to the complexity of product management [129]. Product management that becomes increasingly more difficult as the number of artifacts, e.g. test specifications, increase in number for different versions and variants of a system, which can lead to confusion and faulty use, as experienced in Case 1. Hence, this CPL adds support to the conclusion that any VGT transition project should be driven by a domain and/or test system expert in order to mitigate that unnecessary effort is spent on development of VGT suites from faulty specifications.

In Case 2 another company specific CPL was identified that related to the VGT transition project's limited budget. The limited budget forced the industrial practitioners to develop the first possible VGT script solution they

could identify, which wasn't necessarily the best in terms of script performance, reusability, etc. This CPL is common in industry and affects any process improvement and is therefore not VGT specific.

4.4.1.5 Test system (Environment)

Another CPL that was identified during Case 1 was related to the tested system's simulators. The intended use of the tested system is to control the landing lights and radar equipment at an airport. However, since the actual hardware equipment, e.g. a radar station, is not available during development and testing at the company, the company instead uses a set of different simulators to stimulate the tested system's external interfaces. Hence, simulating the hardware interfaces using software, which is a common practice in industry [65, 130]. These simulators are maintained by the development company and as the tested system has evolved so has the simulators, i.e. new, or additional, functionality has been added to the simulators to better represent the hardware interfaces connected to the system. Additionally, new simulators are constructed over time to test new functionality of the tested system that the previous simulators were not able to test, e.g. timing constraints on the input data which is sent over the external interfaces. However, not all simulators used at the company are compatible with all versions or variants of the tested system. Thus presenting a CPL regarding what test cases that could be automated for the version of the tested system in Case 1 since the test specifications specified the use of newer simulators than the system could support. Therefore, older versions of the simulators had to be used instead. Hence, restricting the usability, reusability and portability of the developed VGT suite as it is currently implemented. The costs required to refactor the VGT suite to work with newer simulators is also unknown and therefore a subject of future work. However, based on the research teams' expert assessment, none of the interactions with the new simulators were determined to be unimplementable. Hence, there is no evidence to suggest that the migration to a new simulator environment would not be possible at the company. This expert conclusion is based on the fact that most of the company's simulators mainly use standard Windows components that Sikuli, through empirical evaluation, has no problem to interact with.

4.4.2 Test tool related CPLs

The CPLs discussed so far have all been related to the tested system, or the development company, and are therefore perceived as CPLs that are plausible to appear in any VGT transition project, with any VGT tool. However, in both Case 1 and Case 2, the VGT tool Sikuli was used and the following section will present specific CPLs identified for that tool. Two sub-categories of CPLs were identified as Sikuli specific, i.e. CPLs related to the tool and CPLs related to the VGT suite that was developed.

4.4.2.1 Test tool (Sikuli) related CPLs

Sikuli is a tool developed and maintained in an open source project that was started at the User Interface Design Group at the Michigan Institute of Tech-

CPL category	**CPL sub-category**	**Nr. of occurrences**	**Percentage of all CPLs**
Test Tool	Test tool (Sikuli)	13	0.224
Test Tool	Test scripts	1	0.0172
	Total	14	0.2412

Table 4.2: *Summary, and distribution, of problems, challenges and limitations (CPLs) related to the test tool (Sikuli).*

nology, but is now maintained and developed by the Sikuli Lab at the University of Colorado Boulder. The tool was at the time of the study still a release candidate, i.e. not a finished product. Consequently, the tool was volatile, containing several defects that affected the stability of the tool's image recognition algorithm, its behavior in general and its integrated development environment (IDE). The defects related to the tool's IDE caused it to crash and freeze sporadically, forcing the user to restart the tool, which, as reported from the researcher who did the data collection in Case 1, and the industrial practitioners in Case 2, added frustration to working with the tool. However, as also reported by the practitioners in Case 2, there are ways of improving the tool's stability, e.g. by running the scripts from the command-line rather than the tool's IDE. No explicit reason was found why this practice raised the robustness of the scripts, but it was perceived, by the industrial practitioners, to be related to defects in the tool's script engine.

During test script development it is often required to terminate the test script execution manually, e.g. due to recognized faulty behavior. However, manual termination of the script execution can cause Sikuli scripts to be corrupted. Hence, a serious CPL, which was identified both in Case 1 and Case 2, with impact on robustness and usability of Sikuli. In the instances where the CPL occurred, it was reported that the script logic could always be recovered but that the links to the sought images in the script all broke, which required recapturing of all the images. Hence, for larger scripts, this CPL can be both time consuming and tedious whilst also adding frustration. The mitigation practice found for this CPL during the project was once again to run the scripts from the command-line rather than from Sikuli's IDE, which made it possible to terminate the script execution without any reported incidents. Thus, the recommendation, given by the industrial practitioners in Case 2, is to restrict the use of Sikili's IDE to script development only. Additionally, this CPL has been identified on both Windows and MacOs versions of the tool, but with perceivably higher frequency in Windows.

The most prominent Sikuli related CPLs are however connected to the tool's image recognition algorithm, e.g. it randomly fails in cases where it has previously succeeded, randomly clicks on generic positions next to the sought images. These random failures were identified in both Case 1 and Case 2 to be the source of the most frustration during the VGT transition since the source of the problem has yet to be identified and appears completely random in nature. Image recognition failure often occurs during script development because the similarity level of the sought image is either too high or too low for

the image recognition algorithm, which uses fuzzy recognition, to find a match, e.g. because there are several similar images on the screen. To ease the script development in these instances, Sikuli's IDE has a built in feature that allows the user to preview what the image recognition algorithm considers a match to the sought image on the tested system's GUI. This feature shows the developer where Sikuli has found a match on the screen, indicated with a square. If several matches are found, these are ranked according to the similarity of the found image compared to the sought image. The ranking is visually displayed with squares of different color, from light blue, for a match of low similarity, to bright red, for a match of high similarity. During script execution, Sikuli will always interact with the image with the highest similarity, if not told otherwise. According to the researcher who did the data collection in Case 1, and the practitioners in Case 2, corroborated by the authors' own experiences, this feature has never failed to find a match, i.e. indicating a 100 percent success rate of the image recognition algorithm. However, it was reported, from both cases, that even though the preview feature indicates that a match, of high similarity, is found in the tested system's GUI, this does not ensure that the image recognition will find the sought image during script execution. Furthermore, even if the link to the sought image has been corrupted in the script, as described above, this feature still shows a match. The successful matching of corrupted images added a lot of confusion and frustration in both Case 1 and Case 2, since this feature's behavior did not align with the behavior of the script, i.e. the feature indicated a match but the script failed.

Another identified CPL related to the image recognition algorithm was that it is often too lenient or too strict on what is considered a match to the sought image. Therefore, a lot of effort during script development is required to fine-tune, i.e. raise or lower, the similarity level of individual images, which in its default setting is set to 70 percent similarity. For larger scripts this practice becomes quite time consuming, especially since no pattern has been identified regarding which images require higher, or lower, similarity level than the default setting. Furthermore, as mentioned above, Sikuli's scripts sometimes get corrupted, requiring images to be recaptured. Recaptured images do not retain the image similarity properties they had before they got corrupted. Hence, the similarity level once again needs to be set manually. Consequently adding further to the frustration to the script development. Furthermore, the need to fine-tune the similarity is considered to be one of the main contributors to development costs, but perceivably also maintenance costs.

Other CPLs related to the image recognition, identified in Case 1, concerns inconsistent ability to find certain bitmap objects, i.e. animated images and images with similar appearance but with slightly different color. The identification of animated images, e.g. blinking or moving images, was required since the tested system in Case 1 had animated buttons that indicated alarms or warnings to the user. For instance, in most airports there are nets that can be raised in order to stop an aircraft in an emergency, e.g. during brake failure. These nets are situated on opposite ends of the runway and should be raised dependent on from which direction the aircraft is approaching the runway. However, in case the wrong net is raised, or both nets are raised at the same time, the GUI warns the user by periodically switching the background color of the buttons, used to raise the nets, from red to yellow and back again in

an interval of roughly one second. Testing this functionality, with confidence, showed to be problematic for Sikuli that often failed to find the buttons, depending on which state they were in. A conclusion can therefore be drawn that Sikuli is limited in its capability to interact with animated GUIs, i.e. lowering the tools usability. However, as mentioned, the tested systems, in both Case 1 and Case 2, were both quite static, i.e. mostly non-animated. Therefore, the CPL did not affect the overall VGT transition but rather just a few test cases.

In addition, the image recognition was, in some instances, unable to distinguish between images with similar but different color and similar appearance. For instance, the tested system in Case 1 is intended to be used both during the day and at night, i.e. during different lighting conditions. Hence, in order to make the GUI more comfortable to use during night, the GUI colors can be switched to a darker color set, effectively dimming down the brightness of the GUI. However, due to to limitations of Sikuli's image recognition algorithm, it was unable to test that that the GUI colors had been changed. The initial test was performed by asserting that the GUI with the lighter color set was not shown after the darker color set had been activated. However, even when the similarity level of the image of the lighter GUI was raised to 99 percent, the image recognition algorithm was still unable to differentiate the brighter from the darker appearance of the GUI. Consequently, the test case could only be partially implemented by identifying parts of the interface that changed more substantially, e.g. buttons that changed color from gray to yellow. However, this test still did not assure, with full confidence, that all aspects of the GUI were changed. Thus, this lack of confidence in the image recognition's capabilities lowers the robustness, usability and portability of the scripts. This CPL may arise when, for instance, a test aims to verify that a button has changed its graphical state after being clicked, given that the images of the different states are similar.

Yet another CPL with Sikuli, experienced in both Case 1 and Case 2, concerns the lack of documentation about Sikuli's scripting language. Sikuli's scripting language, called Sikuli script, is based on Python but has been extended with a set of methods that make use of the image recognition capabilities of the tool. However, as reported by the researcher who did the data collection in Case 1 and the industrial practitioners in Case 2, there is no consistent, searchable, API for all of the methods supported by Sikuli script and what these methods' properties are. As an example, Sikuli script has a method called wait, which originates in Python's sleep function, which delays the script execution for t number of seconds. The method can take a series of different input parameters, i.e. wait(image, t), wait(t, image), wait(image) and wait(t). These input alternatives are not specified in the official Sikuli documentation but all have different behavior. The first alternative, wait(image, t), causes Sikuli to pause the execution for a maximum of t seconds or until it finds the sought image "image". However, if the second alternative, wait(t, image), is used, the script will always pause for t seconds and then try to find the image "image". Both functions are useful in different circumstances, but since the user generally takes the worst case scenario into account when setting the wait time, the second alternative will make the script execution time much longer. This was considered a large CPL in both Case 1 and Case 2 until it was discovered that the wait method had these different capabili-

ties. In addition, this CPL, i.e. the lack of proper documentation for Sikuli script, can discard companies from using the tool. Especially since some of Sikuli script's methods are unintuitive, e.g. the wait method. Hence, this CPL lowers the usability, learnability, reusability, portability and maintainability of the scripts. However, as we identified in our previous work [65], Sikuli script, or Python, is in general an intuitive programming language, even for novice users with limited programming experience.

Both the tested systems, in Case 1 and Case 2, included input areas that accepted swedish words as input. Hence, there were manual test cases that required the user to input swedish words and/or letters, i.e. å, ä, ö. However, Sikuli only supports an english keyboard for typing, i.e. the supported method "type" does not support swedish letters. Thus, in order to use the swedish letters, Sikuli's "paste" method must be used instead, but this method does not work in all instances since it uses the operating systems (OS) clipboard which isn't available, for instance when typing passwords into Windows OS login screen. The solution to this CPL, used by the researcher who did the data collection in Case 1, was to type ascii characters as combinations of pressing the ALT key followed by the ascii code of the letter. Identifying and implementing the solution was both time consuming and added frustration to the researcher. In addition, the fact that Sikuli does not support characters from other languages was puzzling since the tool's IDE can be set to a variety of different languages, including swedish. Consequently, Sikuli scripts have some CPLs in terms of usability, and portability to systems with GUI's in languages other than english.

Similarly, this CPL is present when the tool's optical character recognition (OCR) algorithm is used. The OCR algorithm makes it possible to read texts of bitmap images but, once again, it only supports english letters. Hence, the swedish letter "ä" is interpreted as an "a", "ö" interpreted as an "o", etc. Thus, once again limiting the usability and portability of the scripts.

Consequently, there are many serious CPLs related to the Sikuli tool, some of which are related to the tool's IDE, others to the image recognition algorithm, etc. One cause of these CPLs has been determined to be because of the tool's Java implementation, which, through exploratory experimentation on approximately 50 different computers, was found to be highly dependent on what version of the Java Runtime Environment (JRE) is installed. In addition, the current version of Sikuli, at the time of writing this report, requires Java 6, i.e. JRE 6, and is only stable for certain versions of said JRE. However, no comprehensive evaluation has been performed to find which versions of JRE 6 that make Sikuli more stable, but is a subject of future work. In addition, for users of Java 7, Sikuli's initialization file has to be modified to use the exact path to the JRE 6 executable, rather than the path provided by the operating system's environmental variables. Thus, many users, especially users with limited programming and/or OS knowledge, can be discouraged from using the tool, if they get it to work at all. In addition, our exploratory experiments could also show that Sikuli is more stable, in general, on MacOs than on Windows. Consequently, this CPL limits the robustness, usability, portability and maintainability of the scripts and the tool.

4.4.2.2 Test application

The VGT suites that were developed in Case 1 and Case 2 were both developed as 1-to-1 mappings of the manual test cases. However, the manual tests were, in both cases, designed to be as unambiguous and simple as possible for a human tester to perform. Hence, the test cases were defined in mutually exclusive steps that each consisted of setup of the system, through manual input, to put the system in a specific state before an assertion, i.e. the test, was performed, followed by another setup, input, etc. Thus, the tester would start with test step 1 in test case x and then, without requiring any knowledge of future steps, execute each step of the test case to test step n. The benefit of this approach is that any tester, or even developer, can perform these tests. The drawback is that this approach may be time consuming, since the tester has to jump back and forth between, for instance, a simulator and the tested system several times, i.e. set up the simulator, do the test, do a new setup, etc. However, the test steps, in Case 1, were mutually exclusive, meaning that all the setup of the simulator could have been done in one step rather than several to save time. Grouping all of the simulator setup steps had however made the test case more complex and could potentially have added ambiguity to the test case. However, when executing a test case automatically, ambiguity in the script becomes less of a concern due to the more structured semantics of a script compared to natural language. Furthermore, said complexity is not a problem for the computer, only the human interpreter. Thus, there are other aspects than ambiguity to consider when performing automated testing, which are more important such as script performance, quality and reusability. Consequently, a 1-to-1 mapping between manual tests and VGT scripts is not necessarily the best approach. On the one hand, the 1-to-1 mapping approach allows the scripts to be verified through comparison with manual test execution, but on the other, it can have negative effects on the performance of the script. An alternative automation approach is therefore to group all related interactions with similar GUI component in one test step, given that they are mutually exclusive and do not affect the flow of the test scenario. The benefit is that the execution time becomes lower, whilst still allowing the developer to verify the test script outcome with the manual test case. The CPL lies in identifying these mutually exclusive test steps and group them together, correctly, in the script. Furthermore, this practice contains a trade-off since it raises the maintenance costs of the scripts because changes to the manual test cases become harder to update in the scripts. Another potential automation approach is to disregard the manual tests all together. Hence, instead of using the manual tests as a specification use domain expertise to build an automated test suite for the core functionality of the system. The drawback of this approach is that it requires domain and system experts to write the scripts, which might be a CPL due to the associated cost. An alternate, inverted, approach is to only automate the test suite's large and complex test cases, e.g. test cases that are prone to faulty execution by a human, or test cases that are so long that they become cumbersome for a human to execute. These alternative automation approaches are supported by information acquired from the practitioners in Case 2 that in the future will focus on developing a more generic VGT application to test all the basic functionality of the tested system,

CPL category	**CPL sub-category**	**Nr. of occurrences**	**Percentage of all CPLs**
Support software	Third party software	10	0.172
	Total	10	0.172

Table 4.3: *Summary, and distribution, of problems, challenges and limitations (CPLs) related to the support software, e.g. VNC.*

i.e. not following the manual test specification.

Another, more concrete, CPL, related to the VGT suite, which was identified in Case 1, consisted of a combination of how Sikuli uses the mouse cursor and the speed of the tool. In order to mark and copy a generic text from an application, a human can double-click one the text and then copy it using a keyboard shortcut. This functionality was required in some of the test cases in Case 1. However, since Sikuli performs the double-click with such high speed, i.e. much quicker than a human, the operating system did not always register both of the clicks, which caused the script to fail. This CPL is minor, and can be solved by changing Sikuli's settings to lengthen the time between clicks. However, the CPL is still worth mentioning because even though Sikuli, as all other VGT tools, interacts with the tested system in the same way as a human, it is not human. Thus, the developer needs to consider what Sikuli is actually doing, underneath the hood, when the script is being developed and executed to avoid CPLs that originate from Sikuli acting "non-human".

An alternative solution to the text marking CPL, discussed above, which was found to be more robust, is to search for text using different applications search functions, e.g. in text editors. The search function was used in Case 1 in test cases that required XML files to be rewritten in order to change the layout of the tested system's GUI, since a found match to a search, generally, automatically marks the found text. The drawback of this approach is that you have to know what text you are looking for, and it only works in systems that have a search function. A third alternative solution is to use Sikuli's Optical Character Recognition (OCR) algorithm that allows the tool to transform text in images to strings that can be used for further processing in the script. However, Sikuli's OCR algorithm was found to be unreliable in the available version of the tool, at the time the study was performed, and was therefore used as sparsely as possible.

4.4.3 Support software related CPLs

As shown in Figure 4.2 the test system in Case 1, included several computers connected through LAN, which Sikuli interacted with using a third party virtual network connection (VNC) application. A similar setup was used in Case 2, but only between two computers rather than four. The reason for using VNC was two-fold. First to allow Sikuli to perform distributed test cases, i.e. test cases that required interaction on different computers. Second to make the Sikuli's script execution non-intrusive, i.e. removing the impact of running the performance intensive image recognition on the same computer as the tested

system. Non-intrusiveness helps mitigate the CPL that the image recognition algorithm might steal computational resources from the tested system, which could potentially change the tested system's behavior during runtime, i.e. slow down the execution. Thus, put the system in states that will not occur during real-world use or cause the test scripts to fail due to failed synchronization between the scripts the tested system.

However, the use of VNC was also the cause of several CPLs. First, when Sikuli is executed locally it can remove the mouse-pointer from the screen, making the mouse-pointers location irrelevant for the success of an image recognition. However, when Sikuli is executed over the VNC application, the mouse pointer cannot be removed since it is rendered remotely on the target computer. The mouse pointer can therefore obstruct buttons, and other sought bitmaps, thereby causing the image recognition to fail. Furthermore, the use of VNC had degenerative effects on the success rate of the image recognition because of network latency. Even when the VNC application was running in an optimized setup, the frame rate of the viewer caused the image recognition to fail. This CPL was especially troublesome for test cases that included interaction with animated graphical components, e.g. the emergency nets in the tested system in Case 1, since the lower frame-rate could cause the buttons to be distorted as they were toggling color. Hence, on occasion, the buttons were rendered with the top half of the button in red color, and the button half in yellow color, or vice versa. In addition, VNC introduced other, less obvious, faulty behavior. The increased faulty behavior was observed in both Case 1 and Case 2, but VNC was identified as the source of the CPLs in Case 2. Hence, even though running Sikuli over VNC increases the tool's usability, it also lowers the tool's robustness since this practice increases the chance of image recognition failure.

In addition, as found in Case 1, the choice of VNC application is a relevant factor. At the start of the project, a more simplistic VNC application was used, but it was soon discarded due to poor performance and because it was unable to send keyboard commands to the tested system. These keyboard commands, e.g. CTRL+ALT+DELETE and CTRL+V, where used to simplify, and/or, where required, to perform some of the test cases. Another VNC application was therefore acquired, which solved the CPL regarding the keyboard commands and also increased the stability and speed of the remote image transfer. Consequently, even though VGT tools can interact with any third party software, the developer should, if there are multiple software options, seek to find the one most compatible with VGT.

Furthermore, the VNC application sometimes lost its connection, which made the screen freeze or caused the application to minimize, which caused the scripts to fail as well. This CPL was experienced in both Case 1 and 2, but no solution was found to resolve it. In addition, as also identified in both cases, the VNC application sometimes distorted the colors of the SUT's GUI, typically during start-up. This CPL was solved in Case 1 by adding script functionality that restarted the VNC application when the distortion appeared.

In Case 1 a third party recording software, Camtasia, was, as mentioned, added to the VGT suite application. The tool was used to capture recordings of the test suite execution which can help developers to identify the cause of

Tier 1 CPL category	**Tier 2 CPL category**	**Nr. of occurrences**	**Percentage of all CPLs**
Test System	Test System version	20	0.344
Test Tool	Test tool (Sikuli)	13	0.224
Support software	Third party software	10	0.172
Test System	Test System (General)	6	0.103
Test System	Test System (Defects)	6	0.103
Test System	Company specific	1	0.0172
Test System	Test System (Environment)	1	0.0172
Test Tool	Test scripts	1	0.0172
	Total	58	∼1 (0.998)

Table 4.4: *Summary of distribution of problems, challenges and limitations (CPLs) that were identified during the automation process ordered according to occurrence of the Tier 2 CPLs. Tier 1 CPL-categories have been listed for each sub-category.*

faults in the system and recreate the faults. However, during the project, in several occasions the software could not be started during script execution, which resulted in the VGT suite terminating before all the test cases had been executed. This CPL shows that even though VGT is able to interact with any bitmap component on the screen, precautions still have to be taken that said interaction is robust, in all aspects of the developed VGT suite.

4.4.4 CPL Summary

58 challenges, problems and limitations (CPLs) where identified during this project, primarily through analysis of the information collected in Case 1, at Saab in Gothenburg, corroborated by information provided from Case 2, i.e. Saab in Järfälla. In the analysis the CPLs were categorized into three tiers, with the lowest, Tier 3, containing 29 mutually exclusive groups of CPLs, as shown in Figure 4.4. Table 4.4 summarizes, numerically, how these 29 groups were divided over the Tier 2 CPL categories, and in turn how the Tier 2 CPLs were divided over the top three Tier 1 CPLs. The top three CPLs concern either the tested system itself, the test tool or support software not directly connected to the tested system itself, e.g. the third party virtual network control (VNC) software. Analysis of the collected CPLs shows that most of them relate to the tested system itself, rather than the testing tool, i.e. Sikuli. However, the most prominent CPLs were determined to concern the tool and its image recognition algorithm, which sometimes failed unexpectedly, had limited ability to interact with animated graphical GUI bitmaps, etc. In Table 4.5, the eight most prominent CPLs have been summarized together with their impact on VGT's applicability in industry or the quality of a VGT suite. These eight CPLs were chosen based on occurrence during the project, but also more subjective measures such as added frustration, confusion, etc.

Nr	Title	Description	Q-attr.	ID. at
1	Img. rec. volatility	Image recognition randomly fails for reasons unknown causing failures in scripts that previously worked. Assumed to be related to Sikuli's immaturity.	Rob, Usa.	C1 and C2
2	Img. rec. limitations	The image recognition algorithm has limited ability in finding certain animated objects and differentiating between objects with similar color, e.g. dark to light gray.	Rob, Usa.	C1
3	Negative VNC effects	Running Sikuli over VNC enables distributed system testing. However, image recognition failure increases due to color changes, mouse cursor placement, etc.	Rob, Usa, Port.	C1 and C2
4	Sikuli IDE volatility	Sikuli is not a finished product and therefore has faults that cause it to crash, loose links to images in scripts, fail to start, etc. Note that these CPLs do not include image recognition related CPLs.	Rob, Usa, Port, Reu, Main, Mod.	Case 1 and Case 2
5	1-to-1 mapping	Using manual test cases as the specification for the automated scripts is not always feasible nor appropriate in terms of performance, maintainability, etc. Furthermore this documentation can be faulty.	Usa, Port, Reu, Main.	C1 and C2
6	Test system limitations	Test system CPLs are context dependent. A reoccurring CPL is however synchronization between scripts and the tested system. Especially in web systems.	Rob, Usa, Reu, Port, Main.	C1 and C2
7	Hardware limitations	Image recognition is a performance heavy, dependent on hardware support. Especially in real-time systems with animated interfaces.	Rob, Usa, Reu, Port.	C1
8	Test system maturity	Expected output imagery requires the system to have reached a suitable level of maturity.	Rob, Usa, Reu, Port, Main.	C1 and C2

Table 4.5: *A summary of the eight most prominent identified CPLs from Case 1 (**C1**) and Case 2 (**C2**). Prominence is based on occurrence, perceived negative impact on the transition to, or usage of, VGT, added frustration, etc. **Rob** - Robustness, **Usa** - Usability, **Reu** - Reusability, **Learn** - Learnability, **Port** - Portability, **Main** - Maintainability.*

4.4.5 Potential CPL solutions

The focus of this work is on the CPLs related to VGT when performed in industrial practice but four generic solutions were also identified. These solutions have been summarized in Table 4.6 together with the Tier 3 CPLs, from Table 4.5, that they solve or mitigate. The reason for the low number of presented solutions, in this report, is because many of the solutions that were found/used in the study were ad hoc and thereby not generalizable. In Figure 4.5 the four generic solutions have been mapped to the Tier 3 CPLs. As can also be seen from Figure 4.5, not all of the CPLs are listed, which is either because no generic solution was identified to solve them, or because no solution was found at all. However, as shown in Table 4.6, and Figure 4.5 these generic solutions are applicable for solving or mitigation of more than 50 percent of the CPLs.

The first generic solution, used in both projects, was to ensure that the scripts were developed with redundant levels of exception handling to mitigate CPLs such as tool and image recognition volatility, etc. This exception handling was achieved by using Python's inherent exception handling. Developing this exception handling is however quite time consuming and can be complex since these exceptions are most often caused by unexpected tool or system behavior. However, the solution, which was used in both Case 1 and Case 2, was reported as effective.

The failure mitigation in Case 1 consisted of three levels of failure redundancy, i.e. on a method level, on a script level and on a test suite level, as previously described in Section 4.4.1.2. Hence, if the script would fail, the failed interaction method would first be rerun, which if failed again would result in the entire test case be rerun, and finally if that failed as well, the test system would be restarted and the script rerun a third time. For each rerun a textual log of the test execution was automatically produced and for every rerun above the method level, the execution was also recorded. The failure mitigation in Case 2 was less complex, with only one level of redundancy. Hence, if a test case failed, the system would roll back the system, and then continue with the next test case. Rollbacks were in both cases performed with a teardown method similar to the JUnit test framework [11]. This solution is perceived as generic given that the VGT tool which is used for the automation has similar scripting support and solves CPLs such as script failure due to unexpected behavior of the tested system, the operating system or supporting software, or image recognition failure, etc. Some VGT tools also have other types of redundant failure mitigation, for instance several image recognition algorithms and image repair features.

The second solution that was identified, which mitigates the lack of Sikuli documentation, is to continuously document the script development, e.g. document the test suite architecture, the functionality of help scripts and methods. Thus, ensuring that new testers, and/or developers, can more easily start working with the test suite, but more importantly, to mitigate degradation of the VGT suite over time [16]. This solution shows how VGT testing has commonalities with traditional software development. However, documentation is only explicitly required for the test architecture, since the scenario-based scripts are generally intuitive by themselves. This intuitiveness comes from the com-

bination of high level of interaction, which is equal to human user usage of the system, and images in the scripts that define with what these interactions are performed. In addition, given that the scripts are implemented as 1-to-1 representations of the manual test cases, the manual test case descriptions also serve as specifications, and documentation, for the test scripts.

The third identified solution regards the removal, or non-usage, of remote computer control software, e.g. virtual network connection (VNC) applications. Removal of VNC from the test architecture raises stability of the test execution by mitigating detrimental effects to the image recognition due to network latency, lowered frame-rate, etc. However, this practice is a double-edged sword, because, even though it raises robustness, it also restricts the number of test cases that can be performed on distributed systems. In addition, by running the test scripts locally, more load is put on the computer running the tested system. Hence, raising the risk of faulty behavior of the tested system due to the lack of performance resources, e.g. access to the central processing unit (CPU) or the computers random access memory (RAM). However, as reported by the practitioners in Case 2, by running the test suite locally, a success-rate of close to 100 percent could be achieved, whilst with VNC, in their context, only a success-rate of 70 percent could be achieved. Consequently, the use of VNC allows VGT tools, e.g. Sikuli, to automate manual test cases for distributed systems, but potentially also lowers the success-rate of the image recognition algorithm.

The fourth solution aims to solve the CPL that VGT scripts generally execute quicker than the tested system can update its GUI, i.e. the scripts are not properly synchronized with the tested system. This CPL was reported, in both Case 1 and Case 2, to be a huge source of frustration during the automation and also very time consuming since no explicit pattern could be identified when and where delays had to be added in the scripts to synchronize them with the tested system. However, as reported by the researcher who did the data collection and industrial practitioners, this CPL could be mitigated through systematic insertion of delays in the scripts at locations which could, later during the project, be estimated upfront based on experience of working with the tool. In Case 1, this solution was also supported by the development of custom methods with an additional time delay parameter. For instance, the click(img) method from Sikuli's instruction set, where "img" would be the sought image, was expanded to create a click(img, delay) method which delayed the script execution "delay" number of seconds before performing the click. These custom methods provided additional robustness to the scripts but also increased execution time since these methods required the sought image to be found twice, first by Sikuli's wait for image function and second by the click function, i.e. doubling the minimum number of required image recognition sweeps. However, due to the increased robustness, the researcher who did the data collection in Case 1 reported that it was still beneficial, especially since the image recognition algorithm in Sikuli is quite fast, i.e. can perform upwards of 5 complete image recognition sweeps of the computer monitor per second. This solution is proposed as generic since most VGT tools provide methods that can wait for the system to reach a stable state before the execution continues.

Finally, even though several generic solutions and mitigation practices were

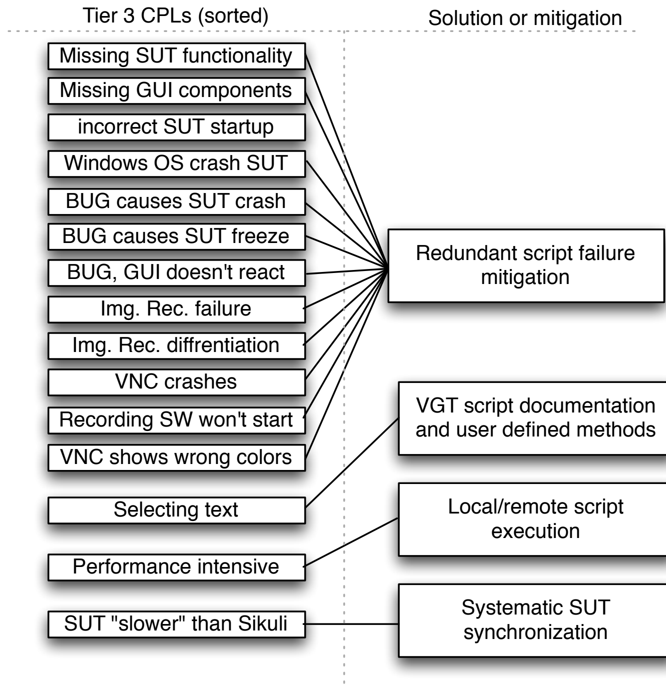

Figure 4.5: *A visualization of how the identified, potential, solutions and mitigation practices, summarized in Table 4.6, connect to the 29 mutually exclusive Tier 3 CPLs.* ***SUT*** *- System under test,* ***VNC*** *- Virtual Network Connection,* ***Img. Rec.*** *- Image recognition,* ***GUI*** *- Graphical User Interface,* ***OS*** *- Operating system,* ***SW*** *- Software,* ***Func.*** *- Functionality.*

found for the CPLs, there were still CPLs that required ad hoc solutions. These solutions were specific to the two projects and could therefore not be generalized. Some of these solutions have been mentioned in Section 4.4. In addition, there were some CPLs that could not be solved or mitigated because they required larger effort to be solved and where therefore out of scope for this study, e.g. changes to the development company's documentation process, and were therefore out of scope for this project. The unsolvable CPLs can be grouped into two categories. First, CPLs for which potential solutions could be identified, but which required so much effort that they were out of scope for the project. Second, CPLs where no solution was identified at all, e.g. how to ensure alignment between the test specification and the tested system.

4.4.6 Defect finding ability, development cost and return on investment (ROI)

Thus far, this report has focused on the CPLs related to the transition and usage of VGT for high system level test automation, and the number of CPLs have been considerate. However, the industrial practitioners in Case 2, still stated that VGT, performed with Sikuli, is a valuable and cost effective technique. They reported that they did not encounter any functionality that they

Nr	Title	Description	Solves problem
1	Script failure mitigation	Failure mitigation code, e.g. exception handling, mitigates tool and image recognition volatility and failure due to unexpected system behavior.	Image recognition volatility and limitations, Negative VNC effects, test system limitations.
2	Script documentation and custom methods	Script robustness, usability, reusability, etc., can be improved by custom methods and reusable artifacts. Additionally, all scripts shall be documented to ease new development and maintenance.	Sikuli IDE volatility, 1-to-1 mapping.
3	Local/ Remote execution	Mitigated use of VNC raises script execution stability but limits use for distributed systems.	Negative VNC effects, 1-to-1 mapping.
4	Systematic synchronization	VGT scripts must be synchronized with the tested system's execution. Custom methods with smart usage of the "wait for image method" in Sikuli can facilitate systematic synchronization.	1-to-1 mapping, test system limitations.
5	Other or no solution	CPLs often require ad hoc solutions. Solutions such as: analysis and replacement of the manual test specifications, hardware and software reboots, change of VNC application, partially implemented test cases, etc. For the sake of completeness, we also state here that there were CPLs that could not be solved, e.g. the missing system functionality, stopping Sikuli from getting corrupted, etc. Hence, there are CPLs that should be investigated in future work to be solved and/or mitigated.	Image recognition volatility, Image recognition limitations, Negative VNC effects, Sikuli IDE volatility, test system limitations, Hardware limitations and test system maturity.

Table 4.6: *A summary of general solutions that were identified for the CPLs listed in Table 4.5.*

could not automate using Sikuli, only that it was more or less challenging. However, since VGT is a test technique, the primary measure of successful application is still the defect finding ability of the developed VGT suites. For the tested system in Case 1, six different defects were identified.

1. Switching between military and civil landing lights did not work.
2. The button to switch military and civil landing lights did not disable as intended.
3. The button to switch military and civil landing lights did not disappear as intended.
4. Switching quickly between runways caused the tested system to freeze.
5. Tabs for switching between views would not load.
6. Logging out of Windows caused the tested system to freeze (The main thread of the tested system would not terminate).

These defects were reported in detail to the company and have since the study been corrected and are no longer present in any commercial version or variant of the system.

These defects were identified during implementation or execution of the VGT suite. Analysis of the found defects found that four of the defects were previously known to the company and already corrected in later versions of the system, whilst two were still unknown. Further analysis showed that the manual test cases could identify all six defects, but since two of the defects were sporadic it required several tries to force their faulty behavior and therefore they had not previously been found. Hence, the VGT scripts were able to identify all the defects that the manual tests cases applicable for the tested system could identify, i.e. providing the VGT suite with the same level of confidence, in terms of defect finding ability, as the manual tests.

In Case 2, the industrial practitioners reported similar results, i.e. that their automated tests could identify all the defects that the manual tests could identify. Additionally, as mentioned, they reported that the VGT suite could identify three defects that previously had been unknown to the company. In the past, the manual test cases had only been run once in sequence every development iteration, i.e. starting with test case 1 and ending with test case n. Furthermore, due to budget constraints, not all of the system's manual test cases could be applied each iteration. However, by automating the system tests using VGT, it became possible to run the test cases more often and therefore not only provide higher quicker feedback to the developers, but also cover more test cases. In addition, the automation made it possible to run the test cases in several different orders, which resulted in the VGT suite finding the previously unknown faults. Hence, these results show the importance of what order the individual test cases are executed. Furthermore, since the execution order of the automated test cases is simple to change, VGT allows the user to quickly and cost-effectively cover more meta-level scenarios, at almost no additional cost. However, for this practice to work, the test cases need to be independent, such that they can be run out of order. In addition,

it is important that the teardown, or rollback, methods of each script are well defined to put the system back into known states before the next test case is executed, to mitigate any side-effects of failed tests [11]. Failure to rollback the system can result in testing of invalid states and thereby reporting defects that are false negatives.

Table 4.7 summarizes some of the cost metrics that were acquired in Case 1 and Case 2. Worth mentioning, again, is that Case 1 was performed with only one researcher whilst Case 2 was performed by two industrial practitioners. In addition, the researcher in Case 1 had limited development and testing knowledge for the test system whilst the practitioners in Case 2 were domain experts. Furthermore, only one manual test suite was automated in Case 1, whilst in Case 2 a total of three suites were automated. Two of the test suites in Case 2 were considered to be more complex than the average test suites used at the company, whilst the third was considered equal in complexity compared to the average. Additionally, the manual test suite that was automated in Case 1 was perceived to be of roughly equal complexity as the more complex test suites in Case 2. Here, complexity was evaluated based on the number of test steps of each test case, importance of test case success, complexity of test cases GUI interactions, etc.

Whilst the test suite in Case 1 was built around tables that defined test scenarios with defined input and expected output for each test step, defined on each row of the table, the test cases in Case 2 were built around more loosely coupled use cases. These use cases, examples shown in the right of Figure 4.6, were then tied together on a meta-level to form test scenarios, as shown in the left of Figure 4.6. Figure 4.6, on the left, exemplifies a test chain used in Case 2 which is made up from three different test flows, or scenarios. These scenarios all start with use case 1 and 2, i.e. UC 1 and UC 2, and are then followed by one out of three exchangeable use cases, i.e. UC 3A, 3B or 3C (Middle of Figure 4.6). However, the manual test structure also allowed test scenarios to be of unequal length, exemplified with UC 3AA, bottom left of Figure 4.6. The three different test chains are then joined again, and completed, by UC 4 (Bottom of Figure 4.6). This structure perceivably improved the manual test cases usability, maintainability and reusability since the use cases could simply be switched out in any part of the chain to test newly added functionality of the system. The drawback of this approach, for manual testing, is that many of the test scenarios become very similar and therefore tedious to test, i.e. the tester has to perform the same interactions over and over. However, once automated, the tediousness no longer becomes a problem but the benefits, e.g. reusability and maintainability, are kept intact, since new scripts can easily be formed by reusing the individual use cases together with a newly developed scripts.

Table 4.7 shows the development time of the VGT suites in Case 1 and Case 2. As can be seen from the table, the development time in Case 1 was considerably lower than the development time in Case 2. However, the time in Case 1 is based on very precise measurements, performed by the researcher doing data collection, who reported of the actual time spent on script development in rigorous detail. In contrast, in Case 2 the development time was measured by industrial practitioners in a real-world context where measuring the exact development time was low priority compared to the development

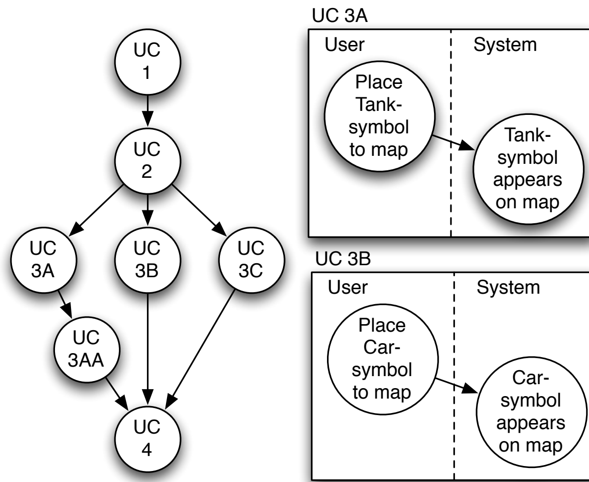

Figure 4.6: *Conceptual example of a test case scenario design, used in Case 2, based on loosely linked use cases (to the right). In the example the test scenario (to the left) contains three unique test-paths, i.e. test cases, that were, prior to the VGT transition, executed manually.* ***UC*** *- Use case.*

Project	**Dev. time (mh)**	**Nr. of Man. test suites**	**Dev. time per suite (mh)**	**Mainte-nance (mh)**	**Manual suite Exe. time (mh)**	**VGT suite Exe. time (h)**	**Pos. ROI after exe-cu-tions**
Case 1	213	1	213	-	16	15	14
Case 2	1032	3	344	266	80	2.5	13

Table 4.7: *Summary of quantitative metrics collected during the VGT transition projects in Case 1 and Case 2.* ***mh*** *- man-hour,* ***h*** *- hour*

itself. Consequently, the measured time from Case 2 contains more overhead, i.e. time not spent on development, than the measured time in Case 1. Hence, the number of implemented test suites, the manual test architectures, expertise of the script developers, and time measurement methods, all differed between the two cases. Therefore, the data in Table 4.7 concerning development time are from two different contexts and should therefore be treated as such, i.e. not compared directly without taking the context, and measurement methods, into account.

However, Table 4.7 does show some interesting, and comparable, differences in terms of improved execution time between the manual suites and the developed VGT suites. In Case 2, the improved execution time was quite considerable, i.e. by a factor of 16 [2], whilst in Case 1 the improvement was

[2]Manual test execution time was 80 man hours but performed by two testers, i.e. 40 work hours in total.

only marginal, i.e. 1.06. The reason for this difference can be found in the individual test cases, and what they aim to test. The manual test suite used in Case 1 contains several quite large test cases, test cases that contain loops and tests to verify the safety requirements of the system, i.e. non-functional attributes of the system. The looping test cases, for instance, aimed to verify the functionality of all the buttons of the GUI, i.e. the same test scenario applied to every button, grouped together into one test case to save space. Performing these tests manually is very time consuming, and tedious, and are therefore often only partially performed, i.e. only a few buttons are tested during test suite execution. However, with the VGT script all of the buttons can be verified during each execution, providing better coverage, without additional cost. However, these tests, even when performed automatically, take quite some time to execute due to the sheer number of buttons that are tested. Thus, since each loop is measurable in length with the test suite's other test cases but all counted as one single test case, rather than several, it has a large impact on the over execution time of the test suite. Hence, some of the VGT scripts had considerably longer test execution time than others, visualized in a boxplot in Figure 4.7. The boxplot shows that the average execution execution time of the VGT scripts was roughly 27 minutes, with a standard deviation of approximately 44. The cause of this deviation can be found in Figure 4.7, which shows that there were several outliers with much higher execution time, i.e. at most 198 minutes, which also affected the total execution time of the VGT suite. In addition, the execution time of the manual test suite in Case 1, 16 hours, is an ideal time, i.e. when performed by an expert tester who are capable of determining for which buttons it is necessary to repeat all the steps of these looped test cases. For a junior tester, the manual test suite can take upwards of 40 hours, as presented in our previous work [65]. Thus, if the outliers are removed and the calculations are performed with the manual execution time of a junior tester, i.e. 40 hours, the results from Case 1 show that the automated scripts execute approximately 4.5 times faster than the manual tests.

The slow execution time of the automated scripts in Case 1 can also be contributed to the fact that several of the manual test cases required the tester to just sit and wait for a number of seconds, or even minutes, for an event to be triggered. Tests of this nature aim to test, as an example, the alarm notification service in the tested system, i.e. that an alarm is triggered if the tested system looses its connection to its hardware interfaces. As a more concrete example, if the wind measurement service looses its connection to the airports wind measurement sensors, it should wait for the connection to be re-established within x number of seconds. If the connection is not established within that time, an alarm is triggered. Hence, for x seconds, during this test, the tester is expected to just sit and wait, and since the VGT test cases were implemented in a 1-to-1 fashion they also have to wait. These tests have significant impact on the total execution time of the VGT suite and relate back to the previous statement that the VGT tests are unable to execute tests quicker than the tested system can respond. In contrast, the test cases that were automated in Case 2 were more on a functional level and did not consider timing issues, etc, which made it possible for the VGT suite to gain a larger performance advantage over the manual test cases.

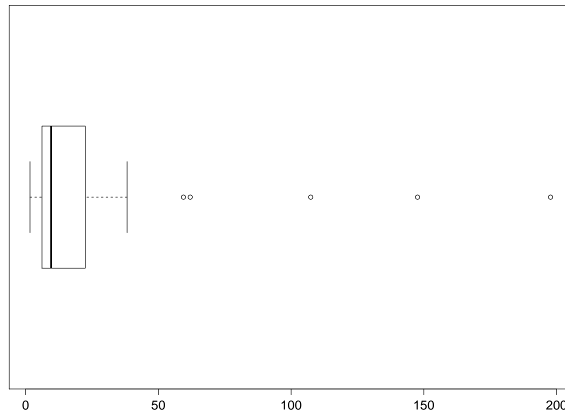

Figure 4.7: *Boxplot showing the execution times for individual test scripts from the VGT suite developed in Case 1. Time, shown on the x-axis, is measured in minutes.*

However, even though the tests were implemented in a 1-to-1 fashion, results from Case 1 and Case 2 showed that the VGT suites had measurably improved execution time compared to the manual test suite execution time. One factor that explains this speedup is that a human tester continuously has to read the manual test steps in order to know what to input and what output to expect from the system. This factor also explains why test experts can execute the test cases faster, since they are familiar with both the tested system and tests and therefore do not need to spend as much overhead time consulting the test specification. However, in contrast to a human tester, the scripts only needs to execute their commands, i.e. removing the reading overhead completely. Additional execution time gains are provided by the scripts ability to quickly paste textual input into the tested system, whilst a human has to write the input. These gains might seem small, but in the overall perspective, i.e. during execution of the entire test suite, these small gains add up and become significant.

Based on the collected metrics, the return on investment (ROI) for the automation can calculated by comparing the development time of the VGT suites and the manual execution time. Once again, these calculated results are not directly comparable between Case 1 and Case 2 if the contexts of these cases are not taken into account. As shown in Table 4.7, the ROI in Case 1 would become positive after 14 executions of the VGT suite, i.e. the cost of executing the manual test suite 14 times equals the implementation cost of the VGT suite. For Case 2, a positive ROI would be reached after 13 executions. However, since there is no cost related to running the VGT suites, and because they can be run at night, they can greatly improve the test frequency and thereby provide daily feedback to the developers [16]. Hence, both VGT suites would reach a positive ROI within one calendar month, if executed every night, including weekends, given that they identify defects.

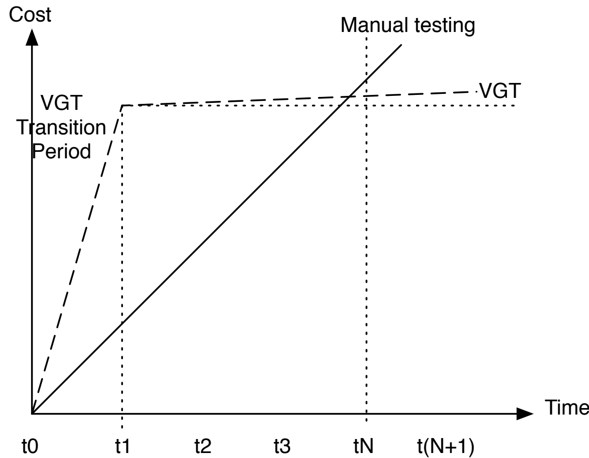

Figure 4.8: *Graph showing a conceptual model of when positive return on investment (ROI) would be reached after the VGT transition in a generic company. The model assumes that each manual test suite execution has a linear cost and that the VGT suite maintenance costs are low.*

This ROI calculation is however simplified, since it does not take the number of found defects into consideration, nor the cost for maintenance of the VGT suite. In addition, the maintenance costs of a VGT suite, developed in Sikuli, are still unknown, only initial data has been acquired, and therefore a more detailed analysis of these costs is an important subject of future work.

A graph visualizing when positive ROI is reached, conceptually, based on VGT transition cost and cost per manual test suite execution, is shown in Figure 4.8, supported by the model defined by Berner et al., 2005 [16]. This model assumes that the cost of the manual test case execution is linear, and that the maintenance costs are small and almost linear. Thus, positive ROI would be reached when the two lines cross, i.e. at time tN in Figure 4.8. It should also be noted that during the VGT transition period, t0 to t1, the VGT suite is not executed, whilst the manual test suite could be.

4.5 Discussion

The following sections discuss the findings from this study, their implications for industry, future work, as well as threats to the validity.

4.5.1 Challenges, Problems, Limitations and Solutions

In this report we have presented challenges, problems and limitations (CPLs) identified in two industrial VGT transitioning projects. 58 CPLs were identified during the study, corroborated by information from both projects. They related to different aspects of the transition or usage of VGT, which we divided into three main groups: problems with the tool itself, the system under test, and the support software or environment.

However, despite the many CPLs, the industrial practitioners from Case 2 reported that they found VGT to be both cost effective and efficient at finding faults. In addition, they reported that even though the CPLs caused frustration during the transition project they found ways, e.g. practices or technical solutions, to mitigate or solve them. The practitioners also reported that they had not found any manual interaction system interaction in any of the system test cases that they could not automate using VGT. Similarly, in Case 1 we found CPLs that were more problematic than others, but reported that many of them could be solved or mitigated.

Together this indicates that most CPLs are solvable in practice with either general solutions like the ones proposed in this report, or ad hoc solutions that solve the CPL in the context where the VGT transition project is performed, e.g. by choosing support software that best fit VGT and the tested SUT. In addition, even though the CPLs had negative implications for the transition or later use of VGT; most of the CPLs were perceived as manageable, which is an important result for future work regarding VGT's long-term industrial applicability. However, before any conclusion can be drawn regarding VGT's long-term applicability, more work is required to determine other aspects of VGT, e.g. the maintenance costs related to the technique. High maintenance costs has been a problem in previous, similar, GUI-based test techniques. Thus, even though VGT is perceived to be able to mitigate many of the limitations of previous techniques, it is not certain that the costs related to VGT are less, which warrants the need for this research.

However, our study also uncovered CPLs that could not be solved because they required process changes at the studied companies, technical improvements of the tool, etc., which required too great an effort to be applicable in this study. What the impact of these unsolved CPLs will have on the developed VGT suites in the future is unknown and therefore a subject for future work. A hypothesis is however that many of the unsolved CPLs will be resolved as the tools, companies and tested systems evolve, either through technical or process development. Furthermore, since several of the unsolved CPLs were related to the companies processes, e.g. lacking documentation practices, and the tested system, e.g. defects in the system, they are perceived to affect any automation technique, i.e. not just VGT. Thus, contributing to the general body of knowledge regarding automated testing.

Furthermore, several CPLs were identified as VGT tool specific, i.e. related to Sikuli, but in contrast to the test system related CPLs, the Sikuli CPLs were reported to be solvable using different practices, e.g. adding redundant failure mitigation in the scripts or running the scripts from commandline rather than the tool's IDE. However, even though most of the tool-related CPLs could be mitigated, some still need to be addressed in the future by further development of the tool, for instance to improve the tool's robustness, make the image recognition algorithm more deterministic, improve the tool's documentation, etc. The tool-related CPLs are of particular importance since they are the most criticized by industrial practitioners, both at Saab and in other companies that have tried Sikuli during our research. Examples of CPLs that have been highly criticized are the tool's unstable IDE, its requirement of what version of Java is installed, the unpredictable behavior of the image recognition algorithm, the limitations of the tool's API, etc. The implications of these CPLs are therefore

that many companies, which could have benefited from the technique, are discouraged from using it. Even though many companies state both a need and want for the technique [67].

However, Sikuli is not the only available VGT tool. In our previous work we have compared Sikuli with an anonymous commercial tool and the commercial tool JAutomate [65, 67]. The results showed that the different tools had different properties that also affect their applicability in different contexts. For instance, whilst Sikuli uses the Python programming language for its scripts, JAutomate has a multi-faceted script environment that has been tailored for both novice and expert users. Thus, Sikuli might be appealing to developers that perform testing, whilst JAutomate is more appealing for testers without programming experience. The tool's different properties also suggests that there might be different CPLs related to different tools, which therefore warrant further studies that replicate the work presented in this report with other tools and in other contexts.

Furthermore, VGT cannot replace manual testing, because VGT scripts, like any other scenario-based script technique, can only find faults specified in said scenarios. Hence, the technique cannot replace an experienced tester's knowledge how to provoke failures and detect defects [16, 42] In addition, even though the technique is perceived to be industrially applicable, there are still several important gaps to cover in the technique's body of knowledge, e.g. empirical evidence to support that VGT's maintenance costs are feasible, before a conclusion can be drawn regarding its long-term applicability.

4.5.2 Defects and performance

In practice, companies are under continuous time-pressure to deliver the software to the customer, which has negative effects on development and quality assurance processes, resulting in software that is likely to contain defects. In the past, one way to uncover these defects has been to use manual high-level practices, which are considered tedious, costly and error-prone [2–7]. High-level automated techniques do exist, e.g. record and replay, but studies have shown that they suffer from several limitations [3, 4, 12, 13, 16, 52]. VGT has properties that perceivably mitigate these previous limitations by, for instance, being robust to GUI layout change, applicable to test applications developed in different languages, etc. Previous work on VGT has also provided initial empirical support for the technique's industrial applicability [65, 121]. However, in order for VGT to also be of value in industrial practice the technique has to be able to identify defects in the tested system. In this report we have presented information, from two projects in industry, which show that a VGT suite is not only able to identify all the defects the manual test suite can identify, but also new ones. These new defects were found because VGT scripts can be executed more cost-effectively compared to manual tests, i.e. potentially every night at minimal cost, with different order between test cases which explores more, potentially faulty, states of the tested system. In combination with the automated high-level interaction the technique can uncover defects that would otherwise have required manual testing, e.g. with new scenarios or exploratory testing, to find. In addition, the execution speed of a VGT suite allows the suite to be executed daily, whilst the industrial norm for manual

system tests is once a month to once a year for complex systems. Hence, a VGT suite can provide feedback to the developers about system-level faults with much higher frequency, which is a requested feature of any automation technique [16]. In addition, due to the low cost of running the VGT suite, it can be rerun several times and execute the individual test cases in different order, thereby cover more meta-level scenarios in the SUT. However, the effects of randomized test case execution is still a subject of future work.

In our previous work, we automated approximately 10 percent of the test cases for another version of the tested system used in Case 1 [65]. Based on the collected data from our previous work, we estimated that all the test cases would take roughly 160 hours to automate. However, based on this new data we can refine the estimate for transitioning the full test suite to 426 hours. This large increase comes from the additional requirements and needs uncovered in this larger study, for continuously running automated test suites the test cases needs to be more robust and there needs to be several layers of redundancy and retries to rule out spurious failures that may not depend on the system under test but on the tool or improper setup of the test environment etc. The estimate has also increased due to a skewed distribution in difficulty where a few unexpectedly problematic test cases was found in Case 1. These problematic test cases for instance included loops, which required more advanced script logic than simple scenarios that started with test step 1 and ended with n. Other problematic test cases required case specific solutions to be found, for instance how to input the Swedish letters å, ä and ö into the Windows login prompt without using Sikuli's type or paste methods. Furthermore, the estimated improved execution time between manual and automated test cases was also incorrect in our previous work. This faulty estimation can once again be contributed to the complex and time consuming test cases that were not covered in our initial study, e.g. test cases that need extensive looping, and test cases requiring the user to just sit and wait for several minutes for a service to timeout. However, based on information collected in Case 2, we found a execution speed-up of a factor 16, compared with the marginal speed-up of 1.06 in Case 1. As reported, the manual test suite in Case 2 only contained smaller test cases, defined in coupled use cases to form longer test scenarios. A conclusion can therefore be drawn that the overall performance increase of a VGT suite is not just dependent on the performance of the SUT, mentioned in Section 4.4.6, but also related to the structure of the manual test specification and how they are carried over into an architecture of auomated tests. Hence, in order to gain the highest performance possible from a VGT suite for a legacy system, one should be selective of which, and in what order, test cases are automated, e.g. not automating test cases that require longer waits or loops first.

Overall we conclude that estimating the time for transitioning a large system test suite to VGT is not an easy task, even though it is likely to be a crucial task in industrial decisions on whether and which test cases to automate. Industrial practitioners should try to sample a diverse set of test cases when doing prototype transitions to get better input to the cost estimation process. Our results also indicate that one should consider using a multiplicative factor of between 2-4 times the bare bone estimates if the test environment cannot be fully controlled or if there VGT tool being used will have to be adapted to

the company and their systems.

4.5.3 Threats to validity

This case study is limited in that only two projects are considered, both in companies working with safety-critical systems of a similar nature and that have been developed and maintained during several years. This is a threat to external validity. In particular the results might not hold for smaller systems or companies that are young or immature in their development practices and processes. However, the companies use fairly different development processes, one more plan-driven and one more agile, but we have found no evidence that this has any large effects on the transitioning to automated test scripts. There was also a difference in the architecture and structure of the test suite at the two companies. We have mitigated this threat by analysing and discussing this difference in detail. Since both projets used the same VGT tool (Sikuli) the results are likely to depend on the tool. However, in earlier research we found that Sikuli had comparable properties and capabilities to a commercial VGT tool [65] so we think this threat is not a major one.

We still caution that the collected quantitative information should not be compared without considering the context of these transitioning projects since the projects were performed with different systems, by individuals with different experience and with different information acquisition processes. However, since data collection procedures differed between projects but still corroborated each other we consider little negative effect from these factors on the identified CPL's; apart from the system-specific CPL's they are likely to be seen also in other contexts. In particluar, since similar results was seen in the projects even though the data collection was very different between projects.

The researcher doing data collection in Case 1 and the industrial practitioner in Case 2 were all inexperienced with VGT and the tool at the start of the projects. We do not consider this a major threat to validity since this is likely to be typical of these types of transition projects in industry. The more experienced researcher can be considered as a kind of expert consultants which would typically be made available in most companies deciding to do large test suite transitioning. They supported data collection and lead study design in early stages. Also, the experienced researchers lead interviews and feedback workshops during the later stages of the project and performed the analysis. Triangulation was also achieved through independent interviews and inclusion of different roles.

There is additional threats to validity in Case 2 since it was driven by industrial pratitioners and the researchers were not on site other than on specific occasions. However, the researchers had continuous contact with the pracitioners and focused on setting up good procedures and communication in early stages, as well as collecting experiences and feedback in later stages. We argue that the threats to internal validity that comes from using industrial practitioners in research are unavoidable. If the empirical software engineering community want data from real, industrial projects a certain lack of control must be accepted. Furthermore, since the industrial practitioners had been tasked with doing the transition and evaluating VGT they were highly motivated to do the work and collect relevant data.

4.6 Conclusions

Software industry is faced with challenges that has created a need for research into high-level test automation, a need that Visual GUI Testing, even though it has many challenges, problems and limitations (CPLs), can potentially help solve.

Many companies in current software industry rely on manual test practices to perform high-level testing, e.g. system and acceptance testing, even though these practices are considered costly, tedious and error prone. Automated test techniques, e.g. unit testing and record and replay, have been proposed as solutions to these problems. However, even though there is work to support the usability of these techniques, there is also empirical evidence to suggest that the techniques have problems that limit their industrial applicability in different contexts. Because of these limitations there is still a need for further research into high-level test automation.

Visual GUI Testing is a technique that is emerging in industrial practice which combines image recognition with scripting to automate high-level tests. Empirical studies have shown the technique's industrial applicability but like any other technique, VGT has challenges, problems and limitations (CPLs). CPLs that have previously not been explored, leaving a gap in VGT's body of knowledge.

In this paper we have presented an empirical study performed in two industrial projects where researchers and industrial practitioners used VGT to automate industrial high-level test cases. During the study, 58 CPLs were identified in total that were categorized into 29 mutually exclusive groups of CPLs that relate either to the transition to, or usage of, VGT in industrial practice. The CPLs were further categorized into eight more generic groups of CPLs that relate to the version of the tested system, the tested system in general, defects in the tested system, the company's processes, the test environment, the VGT tool, the VGT suite or third party software. Further analysis showed that 34 out of the 58 CPLs related to the tested system, the company or the simulator environment. CPLs such as lacking system functionality, misaligned system and test specifications and missing simulator support. Furthermore, 10 CPLs were related to third party software, such as the Virtual Network Connection (VNC) application and screen-recorder software, e.g. VNC lowered the VGT suite success-rate and the recording software wouldn't start. However, the CPLs with the largest impact during the study were 14 identified tool-related CPLs, e.g. unstable tool IDE and unpredictable image recognition behavior, which caused both confusion and frustration among the study participants. The perceived implications of these CPLs are that industrial practitioners may be discouraged from using, or even trying, the technique, whilst also posing concerns for the long-term applicability of VGT in industrial practice, which is still a subject of future work.

Furthermore, the study also identified four generic solutions that would address about half of the identified CPLs, and mitigate their negative effects. Corroborating results from the two projects also indicated that context specific solutions could be found to most CPLs, e.g. development practices or script logic that mitigated the effects of the tool, test system or support software related CPLs. However, in terms of more general solutions there is still a need

for future work, both including technical, e.g. further tool development, and process-oriented solutions, e.g. coding standards.

In addition, the results showed that the researcher who did the data collection and the industrial practitioners found VGT to be both cost-effective and efficient at finding faults despite the CPLs. For instance supported by results from Case 2 where system test frequency was increased from once every six months to several times a week at minimal cost whilst also during the project uncovering three previously unknown faults. Similar results were acquired from Case 1 where four previously known and two unknown faults were identified and reported to the company and corrected. These statements were further supported by the quantitative information that was acquired during the study that indicate that the VGT transition costs are feasible with the potential to provide positive return on investment within one month after development, with up to 16 times quicker execution speed compared to manual tests, whilst still providing equal or even better fault finding ability than manual testsing.

In conclusion, this study has shown that VGT is a valuable and cost-effective technique for high-level test automation but also that it has many CPLs that warrant future research.

Acknowledgment

The authors of this paper would like to thank Saab AB for their participation in these projects and their continued support in answering the question if Visual GUI Testing is an industrially applicable technique.

Chapter 5

Paper D: Maintenance and return on investment

Maintenance of Automated Test Suites in Industry: An Empirical study on Visual GUI Testing

E. Alégroth, R. Feldt, P. Kolström

In submission.

Abstract

The cost of verification and validiation (V&V) is typically between 20 to 50 percent of the total development costs of a software system. Automation is often proposed to lower said cost but there is a lack of empirical data from industrial practice how maintenance affects the cost, which factors affect test script maintenance and what are the maintenance costs involved?

To address this gap we conducted an empirical study with two companies, Siemens and Saab, and evaluate the maintenance costs associated with Visual GUI Testing. 13 factors are observed that affect maintenance, e.g. tester knowledge/experience and test case complexity. Further, statistical analysis shows that developing new test scripts is costlier than maintenance but also that frequent maintenance is less costly than infrequent, big bang maintenance.

Further, a cost model is created to estimate the time to positive return on investment (ROI) of test automation compared to manual testing. We conclude that test automation can lower overall software development costs while having positive effect on quality. In addition, maintenance costs can still be considerable and the less time a company currently spends on testing the longer time they will need to wait for a positive economic ROI of test automation efforts.

5.1 Introduction

The cost of testing is a key challenge in the software industry, reported by both academia and industry to be sometimes upwards of 50 percent of total development cost and rarely below 20 percent [1, 28, 29]. In addition, software industry is moving towards a faster and more agile environment with emphasis on continuous integration, development and deployment to customers [45]. This environment puts new requirements on the speed of testing and presents a need for quicker and more frequent feedback on quality. Often this is used as an argument for more test automation.

Many test automation techniques have been proposed, such as automated unit testing [14,28,32] , property-/widget-based graphical user interface (GUI) testing [55, 119, 120], and Visual GUI Testing (VGT) [50, 65, 121]. However, even though empirical support exists for the techniques use in practice [55, 119–121, 131], less information has been provided on the costs associated with test automation. Further, even if theoretical cost models have been presented, there is a lack of models grounded in actual, empirical data from industrial software systems. Related work [12,16,40,132–136], has also reported on what factors that affect the maintenance of automated testing but not explicitly what factors that affect automated GUI based testing.

In this work we address these gaps in knowledge through an embedded empirical case study [17] with the goal to identify what costs are associated with automated GUI-based testing, here represented by VGT, in industrial practice. First, an interview study at Siemens provides qualitative information regarding the usage and maintenance of VGT from a longer project perspective (Seven months). Second, an empirical case study at Saab the in detail identifies the costs of maintaining VGT suites at different levels of degradation. A heavily degraded test suite is maintained for an industrial system to acquire (worst case) cost information. The maintained suite is then migrated (maintained) to another variant of the industrial system to acquire information about frequent test maintenance (best case). The case study results show that the frequency of maintenance affects the maintenance cost but also that maintenance is less costly than development of the scripts. Statistical analysis on a finer level of script granularity also showed that maintenance of the GUI components the scripts interact with is less costly than maintenance of the test case scenario logic. We also present a correlation analysis that shows that the changes to the manual test cases is a poor estimator for the maintenance costs of the automated test cases.

In addition, observations made during the study support related work [16, 133] that there are several factors, both technical and context dependent, which influence the maintenance. In total, thirteen (13) factors are reported and discussed in terms of their impact on automated GUI based testing. Further, the acquired quantitative metrics are modeled using a theoretical cost model defined in previous work [16, 50]. The model depicts the time spent on VGT maintenance, best and worst case, to be compared to the cost of manual testing at Saab as well as a fictional, but realistic, context where 20 percent of the project development time is spent on testing. From the study we conclude that VGT maintenance provides positive return on investment in industrial practice but is still associated with significant cost that should not be underestimated.

These results provide an important contribution to the body of knowledge on automated testing but also decision support for industrial practitioners that aim to adopt automated GUI based testing.

The continuation of this manuscript is structured as follows. Section 5.2 will present related work, followed by Section 5.1 that will present the research methodology. The paper continues by presenting the acquired results in Section 5.4, which are then discussed in Section 5.5. Finally the paper is concluded in Section 5.6.

5.2 Related work

Manual software testing is associated with problems in practice such as high cost and tediousness and error-proneness [2–7]. Despite these problems, manual testing is still extensively used for system and acceptance testing in industrial practice. One reason is because state-of-practice test automation techniques primarily perform testing on lower levels of system abstraction, e.g. unit testing with JUnit [11]. Attempts to apply the low level techniques for high level testing, e.g. system and acceptance tests, have resulted in complex test cases that are costly and difficult to maintain, presenting a need for high level test automation techniques [3,4,12,13,16,52,97,117]. Another reason for the lack of automation is presented in research as the inability to automate all testing [16, 40, 41, 43, 97]. This inability comes from the inability of scripted test cases to identify defects that are not explicitly asserted, which infers a need for, at least some level of, manual or exploratory testing [116].

GUI-level testing can be divided into three chronologically defined generations. The first and second generation techniques are performed either by capturing the exact coordinates (first generation) where a user interacts with the system under test (SUT) on the screen or through the GUI components' properties (second generation) [6, 106, 107]. However, first generation test scripts are sensitive to change of the SUT's GUI layout, even minor changes to the GUI can make entire test suites inoperable, leading to high maintenance costs. Second generation based scripts are more robust and therefore used in industrial practice, e.g. with tools like Selenium [55] or QTP. However, the technique is still sensitive to changes to GUI components and only has limited support for automation of distributed systems and systems built from custom components [12]. These limitations originate in the technique's approach of interacting and asserting the GUI model rather than the GUI shown to the user on the computer monitor. This approach requires the tools' to have access to the SUT's GUI library or other hooks into the SUT, which limits the tools use for SUT's written in certain programming languages and/or certain GUI libraries. Consequently, second generation tools are considered robust but unflexible.

The third generation, also referred to as Visual GUI Testing (VGT) [50, 65, 121], instead uses image recognition that allows VGT tools, e.g. Sikuli [20] or JAutomate [67], to interact with any GUI component shown to the user on the computer monitor. As a consequence, VGT has a high degree of flexibility and can be used on any system regardless of programming language or even platform. Combined with scenario-based scripts, the image recognition allows

the user to write testware applications that can emulate human user interaction with the SUT. Previous research has shown that VGT is applicable in practice for the automation of manual system tests [50, 65, 121]. However, only limited information has been acquired regarding the maintenance costs associated with the technique [50, 121, 131].

Test maintenance is often mentioned in related work on automated testing but empirical data on maintenance costs from real, industrial projects are limited. There is also a lack of cost models based on such data and what factors that affect the maintenance costs. In their systematic review on the benefits and limitations of automated testing, Rafi et al. only identified four papers that presented test maintenance costs as a problem [40], yet only one of these papers addressed cost and then in the form of theoretical models [16]. Empirical papers exits that report maintenance costs but for open source software, e.g. [58, 131, 136], but the number of papers with maintenance costs of automated test techniques on industrial systems are limited, e.g. [12, 50, 65, 121]. Whilst our previous work reports that VGT can be applied in practice [65,121] but that there are challenges [50], Sjösten-Andersson and Pareto reports the problems of using second generation tests in practice [12]. Instead, maintenance is mostly discussed theoretically and presented through qualitative observations from industrial projects [16, 133]. For instance, Karhu et al. [133] performed an empirical study where they observed factors that affect the use of test automation in practice, e.g. that maintenance costs must be taken into account and that human factors must be considered, but the paper does not present any quantitative support for these observations. The observations made by Karhu et al. are supported by Berner et al [16] that also proposes a theoretical cost model for maintenance of automated testing, but yet again no empirical quantitative data is presented to support the qualitative observations.

There are many factors that affect the maintenance costs of automated tests, e.g. test design and strategies used for implementation. As reported by Berner et al. [16], design of the test architecture is an important factor that is generally overlooked. Another factor is the lack of architectural documentation of the testware and that few patterns or guidelines exist that promote the implementation of reusable and maintainable tests. Additionally, many companies implement test automation with the wrong expectations and therefore abandon the automation, sometimes after considerable investment [16, 133]. However, not all factors that affect maintenance of automated testing are general and it is likely that not all factors have yet been identified. As such, this work, in association with our previous work [50], contribute to the knowledge about the factors that should be taken into account to lower maintenance costs.

5.3 Methodology

The study's methodology was divided into two phases, as shown in Figure 5.1. VGT maintenance was evaluated at two companies, which were chosen through convenient sampling due to the limited use of VGT in practice. The two companies used different VGT tools, i.e. Sikuli [20] and JAutomate [67],

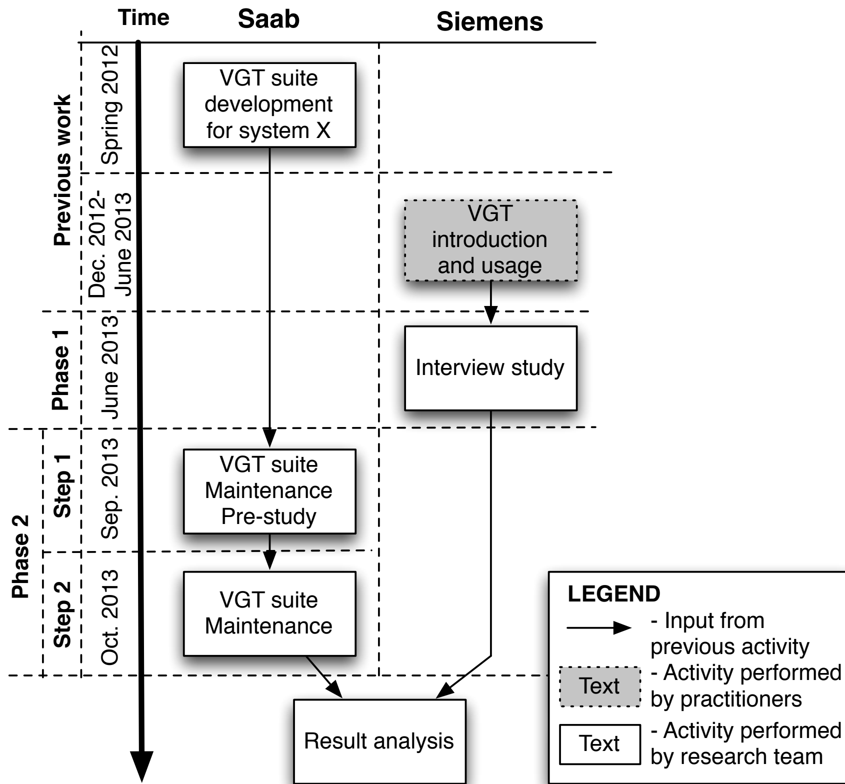

Figure 5.1: *An overview of the methodology used during this work to acquire the study results.*

but, as presented in previous work [65, 67], there is no significant difference between the tools and therefore does not affect the validity of the results.

5.3.1 Phase 1: Interview study

In phase 1, an interview study was performed at Siemens Medical, a company that develops life-critical medical journal systems for nurses and doctors. The studied project was developed by a group of 15 developers and testers working according to an agile development process based on Scrum. Verification of system requirement conformance was performed with unit testing, manual scenario-based and exploratory testing as well as manual acceptance testing with end users. Seven months prior to the study the group had introduced VGT, with the tool JAutomate [67], into their test process in an attempt to lower test cost and raise quality. JAutomate was introduced through development of test scripts as 1-to-1 mappings of existing manual test cases in the project. At the time of the study, approximately 100 out of 500 manual test cases had been automated to be used for continuous integration. However, because of development issues the testers had not been able to make the test suite run automatically from the build server. The scripts were instead started

manually and executed several times a week.

The interviews were exploratory in nature and aimed to elicit the testers' experiences with, and perceptions about, VGT in general and maintenance in particular. Three semi-structured interviews were performed following an interview guide consisting of 35 questions. Each interview was recorded and then manually transcribed and analyzed using a qualitative approach where the answers from each of respondent were triangulated against the other. Discussions were also held with the interviewees after the interviews to clarify and verify the interview results.

5.3.2 Phase 2: Case study Setting

In phase 2, an empirical study was performed at Saab where a VGT test suite for an air-traffic management system, developed in the open-source VGT tool Sikuli [54], was maintained in several steps.

Saab is a developer of safety-critical software with roughly 80 employees in Sweden, split between two development sites, one in Gothenburg and one in Växjö. The study was performed in Växjö where a reference system was made available for the research team for the study. The company develops a set of products using both plan-driven and agile development processes for both domestic and international airports. Most of the company's testing is performed through rigorous manual scenario-based system testing but also automated unit-testing in some projects. The rigorous test process is required for the company's products to be compliant with the RTCA DO-278 quality assurance standard [137].

VGT has been sparsely used at the company through the use of the VGT suite that was created in previous work [50]. The sparse use had however, at the time of the current study, degraded the test suite [16] to a point where it was no longer executable on the SUT. The VGT suite was initially developed as a 1-to-1 mapping of the manual test suite for an older version of the SUT, in the continuation of this work referred to as System X version 1.0. System X was chosen in our previous work due to its complexity, airport runway and radar control, and size, in the order of 100k lines of code. In addition, System X has a shallow graphical user interface (GUI), meaning that interaction with one element of the GUI does not hinder the interaction with any other GUI element on the screen. This GUI property is beneficial for GUI based testing since it reduces the script's logical complexity by, for instance, mitigating the need for synchronization between script and SUT required when opening menus or changing between GUI states to reach the expected output.

System X's VGT suite included 31 test cases, with an average of 306 lines of code per test script (Standard deviation 185) of Python code. Each test case was divided into an average of eight test steps (standard deviation 5) that each contained interactions to set the system in the specific state for verification. As such, the test steps could in another context be considered separate test cases and each test case a test suite due to their size and relative complexity. We emphasize this property of the studied test suite since test case size and structure differs significantly in related work on GUI based testing. For instance, in the work of Leotta et al. [136], where the development and maintenance cost of written and recorded Selenium scripts were compared

for six web systems, each test suite had an average of 32 test cases with a total average of 523 Selenese or 2390 Java lines of code per test suite. To be compared to the approximately 9500 Python lines of code in the studied VGT test suite (not including support scripts) at Saab. The studied VGT suite was developed in a custom test framework, also written in Python, which was created in previous work since Sikuli does not have built in test suite support. A detailed description of the test suite architecture and the framework can be found in [50].

Phase 2 was divided into two steps, as shown in Figure 5.1. In the first step, the manual test specification used to develop the VGT suite for System X version 1.0 was analyzed and compared to the manual test specification of System X version 2.0. In addition, version 1.0 was compared to another variant of the system, meaning another version of the system intended for another customer, we will refer to as System Y, version 2.0. Hence, the study included three VGT suites and three systems, i.e. System X version 1.0, System X version 2.0 and System Y version 2.0, as shown in Figure 5.2. These three systems and test suites were included since they allow to study two different maintenance tasks; one smaller and one larger.

The analysis in step 1 gave insight into the required effort for the maintenance and let us estimate the feasibility of performing the study, as visualized in Figure 5.2. Step 2 then covered a period of two calendar weeks in which the actual VGT maintenance was done for both the smaller and larger maintenance tasks. The two weeks was the amount of time that Saab gave us access to the reference system and Saab personnel.

The analysis in step 1 was performed through manual inspection of each test step by comparing them for all textual test case descriptions between the different test specifications of the three systems. Identified discrepancies were then evaluated to estimate the amount of effort that would be required to maintain/migrate the script. The estimations were performed by a member of the research team with knowledge about both System X, the manual test specifications and the Sikuli testing tool. The required maintenance effort was estimated on a scale from 1 to 10 where 1 was low effort and 10 was high effort. No correctional actions were taken to mitigate estimation bias but the estimations were discussed with practitioners at Saab prior to step 2.

Based on the analysis results, 15 representative test cases were chosen to be maintained in the study. Representativeness was judged based on required effort to ensure that both test cases that required low and high effort were chosen. In addition, the properties of individual test cases were taken into account, for instance if they included loops, branches, required interaction with animated and non-animated GUI components, required support of one or several airport simulators, etc. Basically, the 15 selected test cases were judged to cover the diverse aspects of the test cases and properties while still being possible to maintain in the allotted two-week period.

After the study, the estimated efforts were correlated against the recorded maintenance effort (measured as time) to test the hypothesis:

H_{01}: *It is possible to estimate the effort required to maintain a VGT script through analysis of the changes made to the manual test specification.*

Note that this hypothesis assumes that the VGT scripts are created as 1-to-1

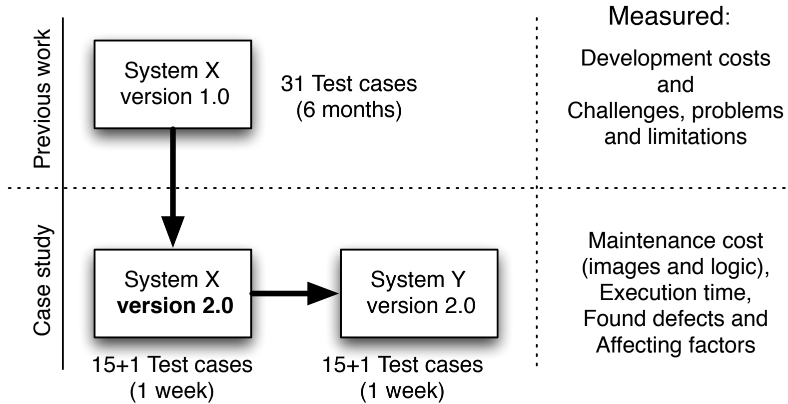

Figure 5.2: *An illustration of the methodology used during the maintenance of the VGT test suite. The Figure shows that the maintenance was first performed in a big bang fashion between System X version 1.0 to 2.0 of the system. Second, System X version 2.0 was maintained for System Y*

mappings of the manual test cases, share test steps, interactions and/or at least have comparable test objectives. Hence, that the automated test architecture is comparable to the scenario specified by the manual test case.

The second step of phase 2 was performed on site at Saab over a period of two calendar weeks (80 work-hours) through hands-on maintenance of the 15 representative test cases that were chosen in step 1. Maintenance is in this context defined as, *the practice of refactoring a test script to ensure its compliance with a new version of its manual test specification and/or to make it executable on a new version of the system under test.*

The maintenance process used during the study followed a structured approach visualized in Figure 5.3. This process was developed for the study since it was observed during previous work that maintenance of a VGT script scenario is error-prone due to the dependencies between different test steps, i.e. test step X is required for test step X+1 to be successful. In longer scripts it therefore becomes difficult for a tester/developer to keep track of all the scenario's steps and the order in which they are executed. The proposed/used process mitigates this problem by breaking the maintenance effort down into smaller pieces that can be individually verified for correctness. However, script verification can require multiple test runs which is tedious and therefore, as we will discuss later, scripts should be kept as short and linear as possible.

The VGT suite maintenance was performed, using pair programming, by one member of the research team and a resource from Saab with expert knowledge about the domain and System X. Pair programming was used because a secondary objective of the study was to further transfer VGT knowledge to Saab.

The maintenance was performed in two parts, as visualized in Figure 5.2. Observant readers will notice that the figure states that there were 15+1 test cases in the test suite. This additional test case was developed during the study based on domain knowledge, rather than the manual test specification, to evaluate if such test cases would be more or less costly to maintain. The

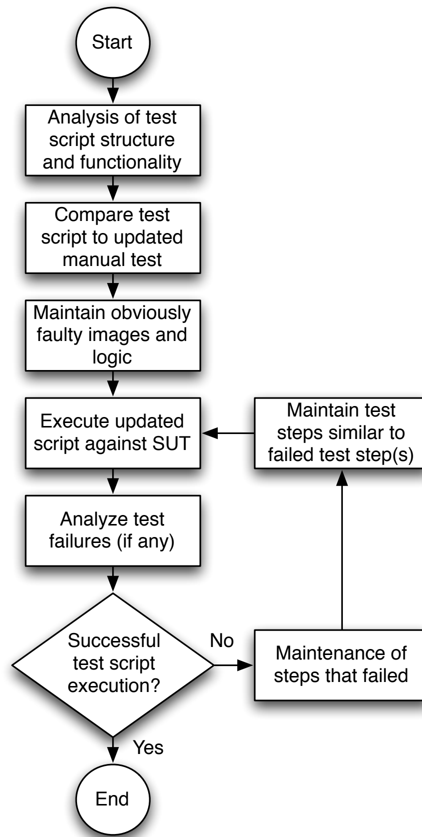

Figure 5.3: *Visualization of the structured maintenance process that was used during the study.*

test case was written early during the maintenance process by the industrial practitioner that was part of the maintenance team and aimed to test each button of a controller for airport TAXI lighting.

5.3.3 Phase 2: Case study Procedure

The first part of phase 2 began by maintaining the 15 chosen test cases from System X version 1.0 to System X version 2.0. In addition, due to changes of System X's functionality and operational environment, substantial effort was required to maintain the VGT test framework itself. Framework changes included refactoring of support scripts and methods, such as fine-tuning of visual toggling between subsystems, support for new simulators, etc. In addition, the tests were refactored by replacing direct image paths in the scripts with variables that were stored in a single support script. Because the support script were of equivalent functionality, complexity and behavior as the test scripts, all qualitative and quantitative metrics were recorded equivalently for all scripts regardless of type, with the exception of execution time. The total

set of maintained scripts were as such 21+1 scripts of which 15+1 were test cases and six were support-related.

In the second part of phase 2, System X was replaced by System Y which was another variant of System X for another airport with another GUI but with minor functional differences. The maintained VGT suite for System X version 2.0 was then maintained for System Y with the purpose of identifying indicative costs associated with regular maintenance of a VGT suite. Hence, a context where the VGT suite had been used and maintained for frequent, or near frequent, integration due to the closer similarities between the different variants of the SUT. The acquired information was used to test the hypothesis:

H_{02}: *There is no significant difference in cost to maintain the VGT suite from version 1.0 to version 2.0 of System X compared to migrating the VGT suite from System X version 2.0 to System Y version 2.0.*

The result of this hypothesis evaluation provides support to the claim that the cost of frequent maintenance, with smaller deltas in terms of changed system functionality, is less costly than big-bang maintenance efforts [16].

The same metrics were collected in both parts of phase 2, results shown in Table 5.2. More explicitly, the metrics that were measured were maintenance cost (time), time spent on maintaining script logic and images, execution time for running the resulting test case, number of found defects as well as qualitative observations, e.g. tool limitations and missing system functionality.

The maintenance cost metric was measured from the start of the maintenance of a new script until it could be executed successfully against System X/Y. Hence, the measured time includes both the time spent on refactoring the script and the time required to verify that the script executed correctly against the SUT. Cost was measured this way to mirror the effort required in a realistic context where test script maintenance has to be followed by verification of test script correctness.

However, the development costs of the VGT suite for System X, version 1.0, collected in our previous work [50], only included the time spent on developing script code. As such, in order to be able to compare the maintenance costs with the development costs, the maintenance costs had to be transformed by removing the average number of test executions for verification during maintenance multiplied with the script execution time. Hence, comparative maintenance time ($T_{X.maint.corrected}$) of a test case was evaluated according the formula,

$$T_{X.maint.corrected} = T_{X.maint.measured} - (\bar{n}_{T.exe} * T_{X.exe})$$

where $T_{X.maint.measured}$ is the true maintenance time including execution time for verification, $\bar{n}_{T.exe}$ is the observed number of average verification runs per test case (constant) and $T_{X.exe}$ is the execution time of the test case. Note that $T_{X.maint.corrected}$ is used for all comparisons between development and maintenance cost in this work but that $T_{X.mait.measured}$ is used in favor of $T_{X.maint.corrected}$ whenever possible. Our reason for limiting the use of $T_{X.maint.corrected}$ is because the number of reruns required to verify the correctness of a script during development or maintenance, $\bar{n}_{T.exe}$, fluctuated between scripts. As such, the constant $\bar{n}_{T.exe}$, equal to the average of required reruns of all scripts, introduces a marginal error in the comparative maintenance value, $T_{X.maint.corrected}$. Since this error presents a threat to validity,

the reader will be informed when $T_{X.maint.corrected}$ has been used in favor of $T_{X.maint.measured}$ and what impact its use might have had on the presented result. Furthermore, since the support scripts were invoked by test scripts during runtime, their execution time could not be measured, as shown in Table 5.2. Therefore, in order to calculate $T_{X.maint.measured}$ for said scripts, $T_{X.exe}$ was set to the average execution time of the scripts in respective VGT suites. Thus introducing an additional error for these scripts.

The quantitative information was then analyzed statistically to test the previously stated hypothesis, H_{02}, and the following null hypotheses.

H_{03}: *There is no significant difference between the cost to maintain images and logic in a script.*

H_{04}: emphThere is no significant difference between the cost to develop and the cost to maintain a VGT suite.

Hypothesis H_{03} was analyzed to evaluate the maintenance costs on a finer level of granularity and is perceived important for GUI based test techniques that use scenario-based scripts that interact with GUI components. GUI components that can be defined as bitmaps as in VGT or as constructs based on the properties of the GUI components as in second generation GUI based test tools. In turn, hypothesis H_{04} aims to investigate if the costs of maintaining the automated scripts are less than the development cost. Since the development costs of the test cases has been shown to be feasible but significant [50, 121], acceptance of H_{04} would infer that automated testing is not feasible.

The collected quantitative metrics were also used as input to a theoretical cost model from previous work [16, 50] to model the total cost of VGT maintenance in comparison to the cost of manual testing performed at Saab. Manual testing that in Saab's context represents seven (7) percent of the total time spent in a software development project. However, as stated, the costs of verification and validation (V&V) in general software engineering practice lies in the bound of 20-50 percent of the total time. The model therefore also includes a plot from a hypothetical but realistic context where the lower bound of 20 percent is spent on V&V. This context is modeled in order to visualize its effects on the time required to reach positive return on investment of test automation. However, it is important to note that decisions to automate are not taken only from a cost perspective; there might be other benefits of automation than its effects on cost.

5.4 Results and Analysis

This section will present the results that were acquired in the embedded study, starting with the quantitative results from Phase 2 that will then be discussed in relation with the qualitative results from Phase 1 and 2. The reason for presenting the results in this order is because the qualitative results help explain the quantitative results, e.g. what factors affected the maintenance effort. Also, note that we in the following section present maintenance cost as time.

5.4.1 Quantitative results

The quantitative results collected during the second phase of the study have been summarized in table 5.2. The table has been divided into three blocks, where the first block (rows 1 to 16) of rows present the data acquired during maintenance of test scripts. In the second block (rows 17-22) the acquired data for the maintenance of support scripts are presented. Finally, the last block (rows 23-24) summarizes the mean values and standard deviation of each column. We once again stress that the development time presented in this table does not take the time required to verify script correctness into account, whilst the maintenance times do.

Analysis of the presented results in table 5.2 show that the maintenance costs for transitioning the VGT suite from System X version 1.0 to System X version 2.0 was higher (mean 110) than the maintenance costs associated with transitioning the VGT suite from System X version 2.0 to System Y version 2.0 (mean 23). Worth noting is that the standard deviation in the first case is almost as high as the average and in the second case almost twice as large. This large deviation is attributed to many factors, such as script size, number of images, number of used simulators, etc. In addition, the deviation is also influenced by the sample being right-skewed above 0. We will return to these factors and their impact in Section 5.4.2.3.

Correlation analysis was used to evaluate H_{01}, i.e. to evaluate if the required maintenance effort can be estimated based on changes to the system's specification. The analysis showed that the correlation between estimated and actual effort between System X 1.0 and System X 2.0 was 0.165 and for System X 2.0 to System Y 2.0 was 0.072. Hence, we reject our hypothesis, H_{01}, that it is possible to estimate the required maintenance effort based on the differences in the manual test specifications for VGT scripts based on said specifications. The reason for the poor estimates relate to the many factors that affect the maintenance cost, which will be presented in Section 5.4.2.3, but which could not be foreseen prior to the study. However, the 15 chosen test cases were still considered a representative sample, supported by the distribution of measured maintenance costs, script functionality, etc.

Further, H_{02} stated that the cost of maintaining the VGT suite for System X version 2.0 would not be statistically significantly different from the cost of maintaining for System Y. Hypothesis testing with the non-parametric Wilcoxon test resulted in a p-value result of 7.735e-05. Hence, we must reject the null hypothesis, H_{02}, at a 0.05 significance level. Analysis of the collected data, shown in Table 5.2, shows that the transition between System X version 2.0 and System Y is lower than the cost for System X 1.0 to System X 2.0. Consequently, since System X version 2.0 and System Y were more similar than System X version 1.0 to System X version 2.0, judged by experts at Saab, this result implies that the maintenance effort is lower if the maintenance is performed frequently rather than big bang. This conclusion also supports previous research into automated testing that stipulate that maintenance should be performed with high frequency to mitigate maintenance cost [16].

H_{03} test if there is any statistically significant difference between the cost of maintaining images and logic in a VGT script. The p-value result from the Wilcoxon test for the maintenance from System X 1.0 to System X 2.0 was

#	Test script	Orig. dev. time (min)	Maint. 1.0X-2.0X (min)	Maint. 2.0X-2.0Y (min)	(logic/ img.) 1.0X-2.0X (min)	(logic/ img.) 2.0X-2.0Y (min)	Exe. 2.0 X (min)	Exe. 2.0 Y (min)
1	t0016	130	100	30	20/80	0/30	3.2	3.25
2	t0017	110	100	35	70/30	0/35	3.167	2.816
3	t0018	265	70	35	35/35	0/35	3.417	3.05
4	t0019	250	230	20	138/92	0/20	3.083	3.067
5	t0014	225	195	15	136.5/ 58.5	7.8/5.2	0.95	4.633
6	t0003	245	120	10	0/120	0/10	0.75	0.783
7	t0024	641	320	65	224/96	19.5/ 45.5	10.5	11.783
8	t0005	705	215	10	107.5/ 107.5	0/10	1.85	2.067
9	t0023	145	315	150	220.5/ 94.5	45/105	9.233	19.4
10	t0007	370	10	5	9/1	0/0	1.783	1.8
11	tS0001	20	20	10	12/8	0/10	1.416	1.416
12	t0026	155	20	5	4/16	0/0	0.333	0.45
13	t0009	40	50	10	10/40	1/9	2.05	1.5
14	t0008	180	35	5	10.5/ 24.5	0/0	4.483	4.5
15	t0041	140	35	20	10.5/ 24.5	0/20	4.617	4.517
16	t0037	140	120	20	84/36	0/20	7.183	7.617
17	vncS.	415	250	50	175/75	35/15	N/A	N/A
18	SimS.	30	40	0	28/12	0/0	N/A	N/A
19	sysS.	370	15	0	6/9	0/0	N/A	N/A
20	winS.	600	15	0	15/0	0/0	N/A	N/A
21	SimS. t2	70	105	45	84/21	40.5/4.5	N/A	N/A
22	windS.	300	110	0	55/55	0/0	N/A	N/A
23	**Ave.**	**252**	**110**	**23**	**66.1/ 47.1**	**6.763/ 17.009**	**3.626**	**4.541**
24	**Std. Dev.**	**195**	**105**	**37**	**71.634/ 37.485**	**14.353/ 23.838**	**2.989**	**4.866**

Table 5.1: *Summary of the collected metrics for the scripts that were maintained during the case study. The test cases have been listed in the chronological order they were maintained in (denoted tXXXX), whilst support scripts have been listed out of order in the bottom of the list.*

0.8972. Hence, we must accept the null hypothesis, H_{03}, that there was no statistical significance difference between the maintenance required of images and logic at the 0.05 confidence level. This result shows that degraded VGT test suites require maintenance of images to the degree that it is equivalent to the maintenance of script logic. However, for the maintenance of System X version 2.0 to System Y the p-value result of the Wilcoxon test was 0.01439. Hence, we must reject the null hypothesis, H_{03}, that the maintenance cost of logic is not significantly different from the cost of maintaining images. Analysis of the average maintenance costs of images and logic for the two maintenance efforts shows that more effort was spent on updating logic in the maintenance from System X version 1.0 to version 2.0 (66.1 minutes for logic and 47.1 for images) compared to System X version 2.0 to System Y (6.7 minutes for logic and 17 for images). An indication supported by empirical observations from the study where images were found to be easier to update than analysis and refactoring of script code. These observations also support that maintenance should be performed frequently to lower the need to maintain both images and logic at the same time since it is assumed that GUI graphics are changed more often than the underlying functionality of safety-critical software.

Finally, H_{04} test if the cost of maintaining the VGT suites was significantly different from the development cost of the suite. The p-value result of the Wilcoxon test for the maintenance costs for System X version 1.0 to 2.0 compared to the development cost was 0.001868 and for System X version 2.0 to System Y 7.075e-08[1]. Consequently we must reject the null hypothesis, H_{04}, in both cases showing that there is statistical significant difference between the development costs and the maintenance costs. Hence, this result shows that the maintenance costs are lower than the development costs. Furthermore, analysis of the costs, shown in Table 5.2, shows that the average costs are lower than the development cost, in both cases, even when the verification time is included in the maintenance cost. However, the transition between System X version 1.0 and 2.0 was 61.5 percent of the development cost and between System X version 2.0 and System Y 17.5 percent of the development cost of the VGT suite. As such, the maintenance costs of a VGT suite are still significant. We will return to a more detailed discussion about the impact of these results in Sections 5.4.1.1 and 5.5.

VGT is first and foremost a testing technique and as such, in addition to cost, its effectiveness must be judged based on its defect finding ability. During the study, eight defects were found, either during execution or maintenance of the VGT scripts. These defects were found through test cases that were divided evenly across the test suite, i.e. no specific test case found more defects. In addition, these defects were of different type, such as GUI defects, e.g. buttons not working as expected, simulator defects, e.g. misalignment between simulator and system behavior, manual test documentation discrepancies, e.g. faulty test steps, etc. As such providing support to previous work on VGT regarding the technique's defect finding ability [50, 121]. In addition, the spectrum of identified defects indicate that VGT is suitable for system level testing rather than pure GUI testing.

[1] Tested using the approximative value $T_{X.maint.corrected}$.

Figure 5.4: *Plot showing the total cost of development and maintenance in the best and worst case based on the measurements acquired in the study. The graph also shows the cost of manual system testing at Saab and the point where the VGT testing reaches positive return on investment compared to manual testing.*

5.4.1.1 Modeling the cost

In our previous work we presented a theoretical return on investment (ROI) model for the development and maintenance of automated tests [50]. The model depicted that the linearly increasing cost of manual testing would surpass the combined cost of VGT script development and continuous maintenance after a period of time.

In Figure 5.4, the actual, numerical results from phase 2 have been visualized in the proposed model, which shows how the total cost of script development and maintenance grows over time (calendar weeks) for a test suite with 70 test cases, equal to the number of manual scenario based test cases available for the system. Four cost plots have been included in the model with the colors cyan, blue, black and purple. First, the blue line (Short dashes) shows the total cost of development and maintenance if maintenance is performed only once every development iteration (60 weeks) in a big-bang fashion where all test scripts are maintained at once. Second, the purple line (Long dashes with dots) shows the cost of development and regular maintenance of the VGT suite. The line was calculated using the average best case maintenance cost identified in the study with the assumption that at most two scripts would require maintenance each week. This assumption was verified with three prac-

titioners at Saab, at different occasions, who all stated that two tests would be an upper bound due to the slow development rate of their system. Third, the black line (Long dashes) shows the cost of manual testing of System X/System Y at Saab and is based on data from previous development iterations of the system, which is calculated to roughly seven (7) percent of the total project time. The cost of manual testing was developed together with, and verified by, three different practitioners at Saab that had been and/or were responsible for the quality assurance of System X/Y and depicts/assumes the following,

1. Each development iteration includes two complete regression tests of the manual test specification/suite, one at the start and one at the end.

2. Each development iteration includes delta tests based on the manual test specification/suite.

3. It assumes that there is no down period between development iterations where the system is not being developed.

These assumptions, and previous approximations, adds error to the model but based on the verification made by Saab's experts and validity analysis, based on the collected metrics from the study, it is perceived as a valid approximation. Detailed and exact cost data is almost never reliably found in actual, industrial development projects and these types of estimations should be expected even if a detailed time logging system is in place. Finally, the cyan line (Solid) represents the costs of testing in a fictional project where 20 percent of the time is spent on testing. Unlike in the Saab case, the fictional project's testing is the total cost of all test activities, i.e. unit testing, system testing, acceptance testing, etc.

In addition to the costs, the graph also contains three vertical lines that depict the points in time when the automation provides positive return on investment (ROI) compared to manual testing. First, the yellow line (Furthest to the left) Figure 5.4 shows when VGT adoption would provide positive ROI in the fictional project, which is after roughly 45 weeks (approximately 11 months). Second, the orange line (Middle) shows when the adoption of VGT provides positive ROI compared to the manual test costs at Saab if the test suite is maintained regularly, which is after 180 weeks (3.5 years) or 3 iterations of System X/Y. Third, the red line (Furthest to the right) shows when positive ROI would be achieved at Saab if the maintenance is performed in a big-bang fashion, which is after 532 weeks (Approximately 10 years) or roughly 9 iterations.

Note that Figure 5.4 only presents ROI as a measure of cost. However, there are other factors that affect ROI as well, such as the frequency of feedback on system quality to the developers and defect finding ability of the different techniques. As such, the figure only presents a partial ROI model that and does not consider the overall cost benefits for the project such as shortened defect analysis time, raised software quality, etc. These factors are required to acquire a holistic model on the ROI of automated testing but are not further explored in this work.

5.4.2 Qualitative results

This section will present the qualitative results that were acquired in phase 1 and 2 of the study and discuss them in relation to the previously presented quantitative results. We decided to present the results in this order since the qualitative results can more easily be understood when read in light of the quantitative data of phase 2, as reported above.

5.4.2.1 Phase 1: Interview results

In phase 1 an interview study was performed at one division within Siemens Medical that seven months prior to the study introduced VGT with the tool JAutomate. Prior to the introduction of JAutomate all system and acceptance testing had been performed manually by testers or the company's end customers, i.e. nurses and doctors. These tests were performed during a dedicated test phase. *"There is something here at Siemens called a system test phase, the system is tested as a whole when all of the development is completed."* In addition to the manual testing the company also used automated unit and performance testing for lower levels of system abstraction testing. *"For the system, we had unit tests for as much as possible to test the code".* The main reasons for the introduction of JAutomate were to lower test cost and to create a test harness for continuous system testing. *"We just got three machines to run automatic builds on where we are now installing JAutomate". "We will run this at night, JAutomate, after an installation or new build. Then we don't have to run the boring tests that otherwise are manual".* Thus mitigating the need for frequent, costly and tedious manual system testing [2,118].

The tested system was composed of a server and a client application where the server side was covered by a rigorous unit test suite. However, the client side had proved to be difficult to test with unit tests due to its GUI focus, which left a need for another automated test approach. An attempt had been made to cover the GUI testing with the tool Coded test UI but had been unsuccessful because the tool was too costly to work with. *"It took roughly the same time to automate a couple of hundred test cases (with JAutomate) as it took to start building tests in Coded UI (after creating a Coded UI framework)".*

The introduction of JAutomate, including the transition from manual tests to automated JAutomate test cases, took roughly 4 calendar months, following a three month evaluation period. During this time, 100 out of the available 500 manual test cases had been automated and were reported to have a total execution time of roughly four to five hours. Whenever a JAutomate test case was created from a manual test case the company had chosen to remove the manual test from the test plan. *"Yes, we did (remove the manual test cases), but we also wrote many new tests when manual tests didn't exist. Then we only created a JAutomate test".* However, the interviewees still stated that JAutomate should not be considered a replacement for manual testing but rather a complement. *"Complement, because manual tests test other things than a strict script does".* Therefore they still ran the manual test cases during release testing of the system but much less frequently than prior to the adoption of VGT.

When asked about the return on investment (ROI) of the JAutomate adoption, the interviewees were uncertain but they perceived it to have been bene-

ficial for the developed system. *"Hard to say, but we could cut the system test part a great deal which is the important part in this case".*

Furthermore, the interviewees stated that they found the technique beneficial because the VGT tests are more similar to human testing and because the technique is very strict in terms of execution, which contributes to its defect-finding-ability. *"It has the benefit that it repeats it (tests) in the same way. It is powerful that way...a human can not always see the whole picture and react on a discrepancy in that picture. For instance that something is fussy or a discrepant".* In addition, the interviewees found VGT with JAutomate easy to learn and perceived that others could learn it easily given that the user was not technically awkward. To support this claim, the interviewees stated that the company had performed experiments with the system's end users, e.g. nurses, that were asked to write test scripts with JAutomate. However, due to their lack of computer knowledge the experiment had not been successful and a conclusion was drawn that the quality gains in comparison to cost of teaching the end users to use JAutomate were less than having the end users execute the tests manually. *"Maybe they (end user) can learn,..., everyone can learn but they have a longer way to go".* This result partially contradicts the result from our previous work where it was found that technically unsophisticated users could write suitable VGT scripts after one hour of tutoring [65]. However, these results was based on VGT with Sikuli and not JAutomate. Further research is therefore required to evaluate the learnability of the different techniques.

In addition, the interviewees reported that the main problem with JAutomate was the tool's robustness. *'It's the robustness, to get everything to work, to get it to go green (pass) 100 times out of a 100. Also to check that it works here, it is pretty slow".* A problem they reported to be primarily related to timing issues rather than the tool's image recognition. *"You have to spend a large amount of time to handle the programs updates and wait for things and similar, which causes things that worked three weeks ago to stop working."* Timing is a common problem for GUI testing tools, especially for web-based systems where network latency has to be taken into account. The problem originates in how to determine when the tested system has reached a stable state such that it is ready to receive new input. Other issues that were presented were that the script development was slow in comparison to the development of new functionality in the system that made it difficult to get system coverage, tester education to create good tests, structuring the test cases, etc. *"We (the development team) changed quite a lot in a day, but we still feel that their tests worked. It has to be the toughest task there is. To create UI tests to something under development, agile development. We really made changes to everything".*

One of the interviewees also reported that the maintenance could take up to 60 percent of the time spent on JAutomate testing each week, whilst the retaining 40 percent consisted of transition or development of new test cases. Hence, more effort was spent each week on maintaining existing test cases than to write new ones. However, when asked if the amount of required maintenance was feasible, two of the interviewees said yes and the third said no. *"Yes, it (maintenance) is (feasible) since you can get it to run, hopefully, during nightly runs and such, which makes you more secure in the quality of the product and*

development." Furthermore, when asked if the interviewees trusted the tool, one interviewee said no, one said to 95 percent and the last said yes. However, when asked if the interviewees would recommend the tool to other project teams at Siemens, all of them said that they would. In addition all of the interviewees stated that they found the tool very fun to work with and that it was beneficial to the company. However, when asked about JAutomate's worst features the interviewees stated that the robustness, as reported, is low, the tool is slow, that there are logistic problems with large test suites and that the symbiosis with Microsoft testing software could be better.

In summary, the interview results showed that VGT with JAutomate requires a lot of time to be spent on test script maintenance, i.e. up towards 60 percent of the time spent each week with the tool. Furthermore the tool suffers from robustness problems in terms of timing and technical issues when it comes to integration in a build process for continuous integration. However, the tool and the technique are still considered beneficial, valuable, fun and mostly feasible by the practitioners. As such, the interviewees still regarded the benefits of the tool and the technique to outweigh the drawbacks. Hence supporting the results from our previous work with Sikuli at Saab in Järfälla [121]. The acquired results on positive ROI after roughly seven months of working with the technique also support the results from Saab and previous work [50] since the maintenance was still perceived as feasible after this time.

5.4.2.2 Phase 2: Observations

Previous research into VGT has reported that verifying script correctness is frustrating, costly and tedious [50, 121]. Especially for long test cases since verification requires the script developer to observe the script execution and for each failed execution update the script and then rerun it. This observation was also made for maintenance of the scripts, which infers that test scripts should be kept as short and modular as possible. How long a test case should optimally be was not analyzed during this work and is therefore a subject of future research but it is perceived that it is dependent on context.

Furthermore, it was observed that lengthy test cases were also harder to understand and more complex because of lack of overview of the script's behavior. Higher complexity was especially observed in test scripts that contained loops and/or several execution branches. This observation lead us to the conclusion that not only should test cases be kept short but also as linear as possible, a practice that also make the scripts more readable by other developers. Test cases that tested multiple features in one script were also observed as more complex to analyze and therefore more costly to maintain. Hence, the number of features verified per test script should be kept as few as contextually possible. These observations lead us to the conclusion that 1-to-1 mapped test cases, despite the benefit of allowing verification against the manual test specification, are not necessarily the most suitable automation scheme. Especially for scenario-based system and acceptance tests that generally verify many requirements at once, instead 1-to-1 mappings between test cases and requirements are perceived as a better alternative.

To facilitate these best practice, more requirements are put on the test suite architecture to support modular test case design and test partitioning. Reuse

of modular scripts is perceived beneficial to creating several test scripts with incrementally longer test chains since changes to the behavior of the SUT will require all affected scripts to be maintained. Furthermore, script code reuse, as opposed to linking reusable scripts together, makes it tempting to copy and paste script code. Copy and paste was observed as an error-prone practice during the study since the scripts are dependent both on the GUI images as well as the synchronization between script and SUT behavior and should therefore be avoided.

Further it was observed that the correctness of the meta-level/support scripts had direct influence on the success and cost of maintenance due to the reuse of these scripts in the test scripts. Thus, placing more stringent requirements on the robustness of the support scripts. However, script robustness assurance requires additional code that in addition to raising cost also raises script execution time [50]. Thus, supporting the need for frequent maintenance to ensure that the support scripts up to date.

5.4.2.3 Phase 2: Factors that affect the maintenance of VGT scripts

Based on observations from the second phase of the study, triangulated with the interviews from phase 1 and literature, we found 13 factors that impact test script maintenance, summarized in Table 5.4.2.3. Impact was classified into four degrees, which are low, average, high and total. Low impact means that the presence of this factor will lengthen the maintenance time with a maximum of a few minutes, average impact factors will add several minutes up to one hour, high impact factors more than one hour and finally total impact will prevent the test case to be maintained. Each of the factors have been discussed from the context of VGT but generalized to other automated test practices where applicable or perceived applicable.

Number of images: Image maintenance is associated with low cost and is therefore considered low impact. However, the cost increases with the number of images in the script and can therefore become substantial for long scripts since broken images are only found during test script execution. A good practice is therefore to always check if a broken image is reused more than once in the script and/or replace duplicate images with variables if supported by the VGT tool. This factor implies that script maintenance is split into test logic (test scenario) and interaction components (images). Consequently, this factor is perceived common to second generation tools where images are replaced with interaction components that are instead GUI component properties.

Knowledge/Experience: VGT has average/high learnability, as shown in this and previous research [65], due to the high level of interaction that the scripts work with. The knowledge and experience of the user therefore has low impact on script maintenance. However, it is a good practice to have a domain expert perform VGT development and/or maintenance to ease analysis and implementation of domain specific knowledge in the scripts. Knowledge/experience is considered to have a larger impact on automated test techniques that operate on a lower level of system abstraction since they require more technical and domain knowledge.

Mindset: VGT script development/maintenance requires a sequential mindset that differs from traditional programming since the scripted scenarios

Nr	Factor	Description	Impact
1	Nr. of images	Each image that requires maintenance raises cost.	Low
2	Knowledge/ Experience	A VGT script expert can maintain a script quicker than a novice.	Low
3	Mindset	VGT scripting requires a sequential mindset that differs from traditional programming.	Average
4	Variable names and script logic	Common to traditional programming. Better code structure improves readability.	Average
5	Test case similarity	Difference between the system and/or test specification compared to old version.	Average
6	Meta level script	Meta level script functionality adds complexity and is costly to maintain.	Average
7	Test case length	Long test cases are more complex and less readable.	High
8	Loops and flows in the test case	Loops and alternative test flows affect the test scripts' complexity and readability.	High
9	Simulator(s)	Simulators can require special interaction code.	High
10	System functionality	Missing system functionality can hinder test script maintenance.	Total
11	Simulator support	Lacking simulator support can hinder test script maintenance.	Total
12	Defects	Defects in the tested system case hinder test script maintenance.	Total
13	VGT tool	Limitations in the test tool can hinder test script maintenance.	Total

Table 5.2: *Summary of the main factors that were identified that affect the required maintenance effort of a VGT suite. The impact of each factor has been categorized from Low to Total.* Low *impact is defined as a required extra cost to develop a script of a few minutes maximum.* Average *is defined as an increased cost (time) of a few minutes up to an hour.* High *is defined as an increased cost (in time) more than an hour.* Total *means that this factor can completely prevent the script from being maintained.*

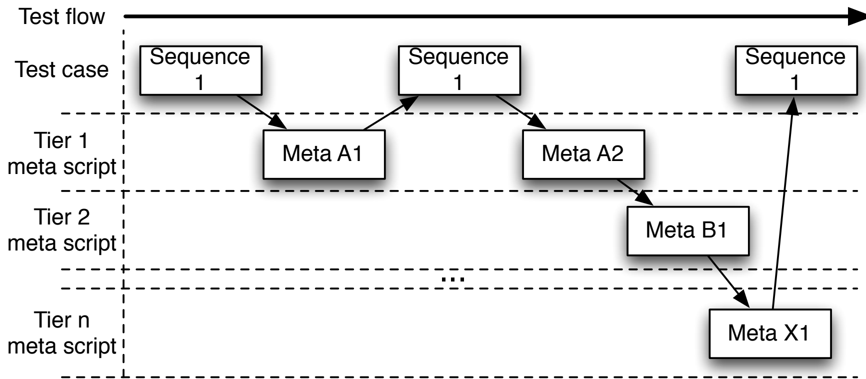

Figure 5.5: *Abstract model of the GUI interaction sequences within a test script, visualizing the complexity of understanding test cases including meta-level scripts.*

need to be synchronized with the timing of the tested system. This factor is considered to have average impact since it is sometimes difficult even for a VGT expert to anticipate when synchronization points are required to align script execution with the tested system's state transitions, which can cause frustration. This factor has been presented in previous work both on VGT and automated testing in general [16,50,133], and to the best of our knowledge there is no mitigation practice. Further, synchronization has been identified as a common problem for automated GUI level test techniques and this factor is therefore considered common to other GUI based test techniques as well.

Variable names and script logic: Common to traditional software development, VGT script complexity impairs script code readability, reusability and maintainability [16] and therefore has average impact on maintenance costs. To mitigate the impact, a good practice is to up front define a clear and consistent test script architecture and define naming conventions for variables and methods. This factor is perceived to be general to all automated testing tools that use more advanced script logic.

Test case similarity: Previous work has shown that test suites degrade if not frequently used and maintained [16], resulting in higher maintenance costs due to increased failure analysis complexity. Degradation is caused by development/maintenance of the tested system and/or changes to the system requirements or manual test cases if the VGT scripts are developed in a 1-to-1 mapping fashion. The impact of this factor is considered average but also general to all automated test techniques. A practice to mitigate the impact of this factor is to frequently maintain the test scripts.

Meta level script: Meta level scripts facilitate test script operation by performing interactions with the tested system's environment that are not part of the test scenario, e.g. start and modify simulators or modify the test system's configuration. Meta level interactions is common in system level tests which put more stringent requirements on the meta level scripts' robustness. Their common use in a test case also lowers test scenario readability, as illustrated in Figure 5.5. As such, this factor is associated with average impact on

maintenance but can be mitigated through up front investment on meta script robustness, e.g. through implementation of additional failure mitigation and exception handling in core scripts. The factor is also considered common to other test techniques and tools that support modular test design where meta level scripts are generally reused to set the system state for certain assertions.

Test case length: Long test scripts are less readable and more complex to maintain because of lack of overview. In addition, longer scripts take longer to execute and verify, which can cause frustration in the case of heavy script degradation. A good practice is therefore to keep test scripts short. Alternatively the scripts should have a modular architecture that allows for subsets of the script to be maintained individually from the rest of the script. Because long scripts are common and not always possible to avoid, the impact of this factor is considered high and also general to other automated test techniques, especially GUI based test automation.

Loops and flows in the test case: Loops and branching test flows should be avoided in VGT scripts because they lower readability and make script failure analysis more complex. As such, a good practice is to keep scripts as linear as possible and break loops and/or branches into individual scripts. Because alternative script flows cannot always be avoided the impact of this factor is considered high and also general to other automated test techniques that support more advanced script logic.

Simulator(s): The purpose of simulators is to emulate software or hardware that will be part of the system's operational environment. Simulators are often crude in terms of GUI design and can be developed in other programming languages than the tested system itself. As such, the impact of having simulators is high for VGT. However, for other automated test techniques it is total due to restrictions of the tools' applicability for certain programming languages and distributed systems, e.g. second generation tools. Hence, this factor is general to automated testing but with varying impact that depends on the the other techniques'/tools' capabilities. The factor can be mitigated for other techniques if the simulators can be operated through simulator APIs.

System functionality: VGT scripts are used for system regression testing and therefore require the tested system to have reached a certain level of implementation to be applicable. Missing or changed functionality can therefore lead to unmaintainable or partially maintained test scripts. As such, the impact of this factor can be total and considered general for techniques that support automated system and acceptance testing. Previous research has shown that the lack of system functionality will limit how much of the test case can be implemented [50]. This factor is considered low for lower level automated test techniques such as unit testing.

Simulator support: Added or changed functionality in the tested system can cause simulators to stop working completely or partially. Partially working simulators can lead to test script failure and inability, or only partial ability, to maintain said test scripts until the simulator itself has been maintained. Thus, the impact of this factor can be total but can be mitigated through frequent maintenance of the tested system's environment. This factor is considered general to all test automation techniques as well as manual testing.

Defects: Defects in the system limits the ability to maintain a test script beyond the test step that finds said defect because further interaction with the

tested system would be in a defective system state that would never appear in practice. Thus, this factor can have total impact on the script maintenance and a good practice is therefore to not execute and only maintain the affect test scripts once the defect in the SUT has been resolved. This factor is considered general for both manual and automated testing.

VGT tool: Different VGT tools have different functionality which make them more or less suitable in different contexts. For instance, only some VGT tools have support to verify that the system can play sound, others have script recording functionality or support for testing of distributed systems. Failure to pick the right tool for the right context can cause this factor to have total impact on both the development and maintenance of test cases. Previous work has compared different VGT tools against each other [65, 67] but how to pick the most suitable VGT tool for a certain context is still a subject of future work. This factor is also perceived to be common to all automated test tools since they all have different capabilities.

5.5 Discussion

The main conclusion of this work is that automated testing, represented by VGT, can provide positive ROI over time when applied in industrial practice despite requiring considerable maintenance. In particular, the results show that the maintenance costs associated with automated test scripts are lower than the development cost of the scripts, shown with statistical significance, independent of if worst or best case maintenance practices are used. Worst case data was acquired empirically through measurement of the costs of maintaining a heavily degraded test suite between two versions of a system at Saab. This was followed by the migration of the maintained test suite to a similar variant of the studied system to acquire approximate but valid data of best case maintenance costs of a VGT suite.

Average maintenance cost in the best case was found to be 23 minutes per test script and 110 minutes in the worst case, whilst the cost of manual execution of a test case was 29 minutes. Plotting the acquired results over time, in a theoretical cost model defined in previous work [16, 50], Figure 5.4 shows that test automation would provide positive ROI at Saab in 180 weeks in the best case and 532 weeks in the worst case compared to manual testing. However, even 180 weeks represent a considerable, long-term, investment to reach positive ROI and the maintenance costs of the acquired test suite would still be significant, upwards of 60 percent of the time spent on test automation each week as reported by Siemens. These results are however placed in Saab's context where approximately seven (7) percent of the total project time is spent on manual testing. Therefore, Figure 5.4 also includes a plot of the verification and validation (V&V) costs in a fictional context where 20 percent of the project time is spent on V&V, which shows that positive ROI is reached in 45 weeks. However, 20 percent is still a lower bound, according to research, of the time spent on V&V in practice that generally spans between 20-50 percent of the total development time of a project [1, 28, 29]. These related results imply that ROI in another context could be reached even faster. Consequently, the time to positive ROI of test automation is directly dependent on the time

spent on V&V at the company prior to automation.

Further, observations from the study show that maintenance of automated tests are dependent on several factors, of which thirteen (13) were identified and presented in this work. These factors include technical factors, e.g. test case length, organizational factors, e.g. the tester's knowledge and experience, environmental factors, e.g. simulator support, etc. Whilst these factors were identified for VGT it is perceived that they are common to other automated test techniques as well, but especially other GUI based test techniques since they, as discussed in previous work, have many commonalities [50]. However, future work is required to verify this statement.

The implications of the presented results and observations are that there are many aspects to consider when adopting test automation in practice. First, test maintenance costs are significant and continuous due to the identified need for frequent test case maintenance to mitigate cost. As such, automated testing will necessarily not lower the time spent on V&V in a project but perceivably lowers the overall project development time by providing frequent feedback regarding the quality of the SUT. This feedback allows the developers to identify defects quicker and thereby mitigate synergy effects between defects that is perceived to lower defect root cause analysis time. Thus, the primary benefit of test automation is raised software quality rather than lowered test related costs. Further, the factors identified in this work that affect test maintenance imply that the technical aspects of the automation, e.g. the test architecture, are key to ensure maintainability of the test scripts. This conclusion in turn implies that best practice of traditional software development should be applied when creating an automated test suite, for instance the test architecture should be modular, there should be code standards for scripting, interfaces to the SUT's environment must be well defined, etc. However, it is not enough to consider the technical aspects, one also needs to consider organizational and human factors such as the knowledge and experience of the tester (tool and domain knowledge), that the tester has a sequential mindset to fulfill the need to synchronize the script execution with the SUT execution, etc. Further support for that these factors' impact on automated testing has been presented in previous work [16, 133] but as the factors presented in this work complement previous work it is uncertain if all factors have been identified. As such, further work is required to identify more factors, what the impact of the different factors are in relation to each other and ways to mitigate the negative effects of said factors.

More specifically, this work also implies that VGT can be applied in industrial practice and provide positive ROI compared to manual testing. Thus partially bridging the gap for a technique for automated system and acceptance testing [50, 65, 121]. It should be noted that the cost data presented in this work does not take the perceived software quality gains provided by more frequent testing and faster feedback to the developers compared to manual regression testing into account. As reported in Section 5.4.1, eight defects were identified during the study either during maintenance or execution of the test suite. The identified defects provide support to previous work regarding VGT's defect finding ability [50, 121]. In addition, the found defects indicate that additional cost savings can be made with VGT through faster defect identification and identification of defects that cannot be found feasi-

bly through manual testing, e.g. defects that appear seldom during runtime. All of the identified defects were reported to the company and has since the study been maintained. However, explicitly how the technique's defect finding ability affects the time to positive ROI is still a subject of future research.

Finally, this work provides a general contribution to the body of knowledge on test automation. Previous work has contributed with empirical results [12, 16,40,132–136] but more is required, especially for GUI based test automation techniques, e.g. VGT, which currently only has limited cost support from industrial practice [121]. Additional empirical support is required to gain a more holistic view on the maintenance costs associated with general automated testing in industry, which requires more studies, in more companies and with different techniques and/or tools.

5.5.1 Threats to validity

One threat to the external validity of this study is that the quantitative data was only acquired for one system and with one VGT tool. Furthermore, the study only evaluated frequent maintenance of the suite in a cross-sectional manner rather than longitudinal. As such, even though the results support that more frequent maintenance is associated with lower cost than big-bang maintenance, more work is required to analyze the costs associated with maintenance in a longitudinal perspective. The results from Siemens indicate that such maintenance is feasible, but more quantitative data is required.

Another threat, related to the internal validity of the results, is that only 21+1 data points could be acquired during the study. However, as stated, the test cases in this work more comprehensive than test cases presented in related work on GUI test automation [136]. Analysis of the test cases indicate that, in terms of size and complexity, each of the test cases could according to previous research be classified as test suites. The average number of test steps per script was seven (7), meaning that the results of this work could be generalized to a context where 147 test cases had been maintained rather than 147 test steps as classified in this work.

Yet another internal validity threat was that only three people were interviewed at Siemens. However, conclusions have only been made on data that could be triangulated from all three sources. Furthermore, since the interviewees were the leading developers on the VGT implementation project it is considered that their statements are credible.

One threat to the construct validity of the study is that the two maintenance efforts were performed on two variants of the same system rather than versions. However, analysis of the system's specifications, supported by statements by the system's developers, show that the differences between System X and System Y are few and therefore our use of the system to measure frequent maintenance is valid.

Another threat was that the VGT maintenance costs had to be measured in a different manner, including test execution time for verification, than the development costs reported in previous work [50]. However, since it was possible to calculate reasonable estimations of the time spent on maintenance programming using the equation in section 5.3.2, this threat is considered low. To verify the output from the equation, the calculated values were discussed

and validated with the leading researcher from the previous study. In addition, the execution time of the scripts is small compared to development and maintenance time of the scripts and as such the observed maintenance costs are still comparable to the development costs without augmentation. Additionally, the augmented data was only used to verify the hypothesis that the development time was not significantly different from the maintenance time, whilst all other statistical analysis was done with the true observations. The reader has also been notified whenever the augmented data has been used in the manuscript.

These threats can partially be explained by being performed in industry under both cost and time constraints that limited the amount of empirical work that could be performed.

5.6 Conclusions

The main conclusion of this work is that automated testing, represented by VGT, will provide positive return on investment (ROI) compared to manual testing in industrial practice. This conclusion was drawn based on data from two different companies, Siemens and Saab. The results show, with significance, that the development costs are greater than the maintenance costs and that the costs of frequent script maintenance are lower than big-bang maintenance. However, the costs of test script maintenance are still substantial, identified at Siemens to constitute upwards of 60 percent of the time spent on test automation each week. A cost that can be compared to the academically reported costs that companies spend on verification and validation (V&V), which range between 20-50 percent of the total time spent in a project [1,28,29]. Thus, the magnitude of required time spent on maintenance is equal to the costs of V&V in general, which infers that test automation may not lower time spent on V&V. However, automated testing can raise the trust in the quality of a system due to more frequent quality feedback that could lower overall project development time. As such, the study shows that the time to reach positive ROI is dependent on the amount of V&V performed by the company prior to automation.

Additionally, thirteen (13) qualitative factors were observed during the study that affect the maintenance costs of VGT scripts but also automated testing in general, e.g. developer knowledge/experience, developer mindset, test case length and test case linearity. The identified factors infer that maintenance of test scripts, but also development, depend on an intricate balance of technical an non-technical aspects in order to reach qualitative tests. Whilst these factors provide a general contribution to the body of knowledge on automated testing, further research is required to identify complementary factors and to measure their impact.

Additionally, this work provides explicit support for the use of VGT in practice by showing that positive ROI can be reached and by providing further support to previous work regarding VGT's defect finding ability. Eight (8) defects were identified during the study, spread among the SUT and its operational environment, e.g. the SUT's simulators. In combination with related work [50, 65, 121], these results show that VGT is a feasible comple-

ment to other manual and automated test techniques in practice to facilitate automated system and acceptance testing.

Chapter 6

Paper E: Long-term use

On the Long-term Use of Visual GUI Testing in Industrial Practice: A Case Study

E. Alégroth, R. Feldt

In submission.

Abstract

Visual GUI Testing (VGT) is a tool-driven technique that uses image recognition to interact with and assert the behavior of a system under test through its pictorial GUI as it is shown to the user. The technique's applicability, e.g. defect-finding ability, and feasibility, e.g. time to positive return on investment, have been studied in previous work. However, there is a lack of studies that evaluate the usefulness and challenges associated with VGT when in long-term use (years) in industrial practice. Such a long-term perspective is generally missing for research in software engineering but in particular for automated testing.

This study bridges this gap through a case study of the company Spotify's experiences of using VGT for several years. Results, acquired through Grounded Theory analysis, show that VGT can be used long-term and has several benefits compared to other test techniques. But it is also associated with several challenges that need to be addressed with organizational changes as well as engineering best practices. In addition, the paper presents a comparison between the benefits and drawbacks of VGT and a custom GUI test approach used at Spotify driven by hooks in the application's source code. The comparison shows that there are several key differences between the two approaches but that they are ultimately complementary for GUI-based testing.

Finally, the paper presents a synthesis of results from the presented, previous and related work that defines a set of guidelines to aid practitioners to adopt and use VGT in industrial practice.

6.1 Introduction

Automated testing has become a de facto standard in software engineering practice, most commonly performed with automated unit tests [14]. Unit testing is performed on a low level of system abstraction to verify that software components adhere to the system under test's (SUT) low level requirements. But unit testing is rarely enough on its own for automated testing in industrial practice; companies also need to continuously verify high-level system properties. The reason is because market advantage is determined by a product's time-to-market in many software engineering domains, which has resulted in a trend that software needs to be developed, tested and delivered as often and quickly as possible. Companies thus want to get human testers out of the loop and automate testing on many levels of system abstraction to reduce costs and increase test frequency. However, the support for automated testing on the GUI level of abstraction is limited, and companies typically complement their low-level, automated test activities with costly manual GUI-based testing [2–4].

We have classified the currently available GUI-level test automation techniques into three chronologically delineated generations [51]. The 1st generation relies on exact screen coordinates, the 2nd on access to the SUT's GUI library or hooks into the SUT and the 3rd on image recognition to stimulate and assert the SUT's behavior. Whilst the 1st generation was deemed unstable and is rarely used anymore, the 2nd generation is commonly used in industrial practice with tool's such as Selenium [55] and QTP [56]. However, the 3rd generation, also referred to as Visual GUI testing (VGT) [64], is currently emerging in industrial practice with tools such as Sikuli [20], JAutomate [67] and EggPlant [68]. Academic research has shown VGT's applicability and some support for its feasibility in practice [64]. However, knowledge from the perspective of long-term use, i.e. use over several years, and what challenges that are associated with the long-term use of the technique, are still missing. The key reason for this lack of knowledge is because VGT has only recently gained a foothold in industrial practice and the number of early adopters that have used the technique for a longer period of time, which can be studied, are therefore few.

However, studies on the long-term perspective of the use of research in industrial practice are generally missing in software engineering [138]. This lack of studies can be contributed to several factors. For instance, such research requires years of investment by a case company, which they are reluctant to invest in areas which have not already been proven. Additional factors include that processes and organizations change over time that cause resources for the research to be diverted. Further, key individuals, e.g. champions, sometimes leave the case companies that cause the research to be abandoned or replaced before the long-term perspective can be analyzed. Thus, research into the long-term effects of a new technique or solution is difficult to achieve. Nevertheless it is important to identify impediments with the research topic to improve its efficacy, efficiency and longevity in practice.

In this paper we address the lack of knowledge on the long-term use of VGT through a single, embedded, case study [71] at the Swedish company

CompanyX[1]. CompanyX is a good candidate given our goals since they have used VGT for several years. They are one of few companies that can provide insights into the entire VGT life-cycle; from adoption to use to long-term use in industrial practice. Here, long-term use also includes the feasibility of maintenance and challenges associated with the long-term use of VGT scripts.

CompanyX develops music streaming applications, for a large number of different platforms and operating systems. These are also continuously updated with new features and functionality that requires frequent regression testing of the application. This regression testing is facilitated by a mature test process which includes automated testing from low-level unit testing to integration testing to GUI-level system testing performed with multiple approaches one of which is VGT. This gives us the opportunity to both study the adoption and long-term use of VGT, its alternatives as well as the development and test context in which it is used.

The study was performed with four (4) interviews with five (5) employees at the company that were carefully chosen to provide a representative view of how automated testing and VGT is used at CompanyX. These interviews were chosen through snowball sampling [98] and analyzed with Grounded Theory analysis. Results were further refined and complemented based on two workshops. Taken together this gives a rich overview of CompanyX's context and answers *(1) how CompanyX adopted VGT, (2) what benefits and (3) challenges the company has experienced with the technique, (4) and finally what alternative techniques the company use for automated GUI-based testing.*

Results of the study show that VGT can be used long-term with several benefits over other test techniques. But there are also many challenges that require both organizational, architectural and process considerations for the technique to be feasible. Because of these challenges, VGT had been abandoned at several projects at CompanyX in favor of a 2nd generation technique. The paper reports the benefits and challenges of the new technique and also compares the technique to VGT based on a set of different properties, including robustness, maintenance costs and flexibility.

The acquired results, together with previous work [50, 121], were then synthesized to create a set of practitioner guidelines for the adoption and use of VGT in practice. These guidelines serve to provide practitioners with decision support for VGT adoption and to prevent practitioners from falling into the pitfalls associated with VGT.

Consequently, this study contributes to the limited body of knowledge on VGT with evidence regarding the technique's long-term applicability as well as much needed practitioner guidelines for adoption, use and long-term use of the technique. In addition, the study provides a concrete, yet limited, contribution to the body of knowledge on software engineering regarding the possibility of long-term use of an academically defined test technique.

The continuation of this paper is structured as follows. Section 6.2 presents related work followed in Section 6.3 of a description of the research process. Section 6.4 then presents the research result, followed by practitioner guidelines for VGT adoption and use in Section 6.5. The results are then discussed in Section 6.6 followed by the paper's conclusion in Section 6.7.

[1] The company's name can not be disclosed due to a non-disclosure agreement.

6.2 Related work

VGT is a tool driven automated test technique where image recognition is used to interact with, and assert, a system's behavior through its pictorial GUI as it is shown to the user in user-emulated system or acceptance tests [64]. Previous empirical work has shown the technique's applicability in practice with improved test frequency compared to manual tests, equal or even greater defect finding ability compared to manual tests, ability to identify infrequent and non-deterministic defects, etc [51, 121, 139]. These studies have also provided initial support regarding the feasibility and maintenance costs associated with the technique, for instance, the maintenance costs of frequent maintenance is lower than infrequent maintenance, positive return of investment can be achieved in reasonable time, the technique can feasibly be used over months at a time, etc. However, several challenges, problems and limitations have also been associated with the use of the technique in practice [50], e.g. lack of script robustness, maintenance related problems, immature tooling and adverse effects caused by the test environment.

Hence, the body of knowledge on VGT covers many perspectives of its use in industrial practice but it still lacks results regarding the technique's long-term use, benefits, drawbacks and challenges, i.e. results that are important to determine the technique's efficacy [138]. However, results on long-term use are difficult to acquire due to VGT's immaturity and limited use in practice that also limits the number of case companies that have used the technique for several years. Thus presenting the need, and importance, of the study reported in this paper.

Related work has evaluated the industrial applicability and feasibility of other GUI-based testing approaches [55,120]. However, the body of knowledge on automated GUI-based testing is lacking empirical results regarding the long-term use of these techniques in practice since most studies only focus on maintenance costs. Maintenance costs that are discussed theoretically and presented through qualitative observations from industrial projects [16, 133]. Some empirical research on maintenance costs have been reported but for open source software [58, 131, 136] whilst the number of papers that include industrial systems are limited [12].

Further, the long-term, empirical, perspective is missing in general in the software engineering body of knowledge, as shown by Höfer and Tichy that conducted a survey of all refereed papers in the Empirical Software Engineering Journal between 1996 and 2006 [138]. Whilst this study is almost a decade old, there is, to the authors' best knowledge, little support to the contrary of this situation today. In their survey, Höfer and Tichy found that empirical research with practitioners is common in software engineering research but that studies that focus on the long-term perspective were missing. A similar observation was made by Marchenko et al. during a three year study with 9 interviews at Nokia Siemens regarding the long-term use of test-driven development (TDD) [140].

A more recent systematic literature review by Rafi et al. also identified that there is a lack of research that focus on the challenges and solutions of automated software testing [97]. In the review, 24,706 papers were surveyed but only 25 reports were found with empirical evidence on benefits and

drawbacks with automated testing. As such, more research is required on the challenges, and further the solutions, with automated testing to improve its adoption and use in practice. [16, 99]. Further research is also required to explore the challenges and solutions from several perspectives that include process and organizational aspects, e.g. how maintenance is performed by testers in practice, and architecture, e.g. how test cases and test architectures are designed [16, 99].

Consequently, this paper addresses several important gaps in the software engineering body of knowledge by supplying empirical evidence of the long-term use of automated testing as well as what challenges, problems and limitations that are associated with the adoption, use and long-term use of automated testing in practice.

6.3 Methodology

The methodology used in this study is based on the guidelines for performing empirical case studies in software engineering defined by Runeson and Höst [17]. These guidelines were used to perform a single, embedded, industrial case study [71] at the software application development company CompanyX.

Objective: The study had two primary objectives. First, to evaluate the long term use and feasibility of VGT at CompanyX, including what benefits and challenges the company had experienced during the adoption and use of the technique. Second, to evaluate the alternative automated techniques used at CompanyX for GUI-based testing.

Unit of Analysis: As such, the unit of analysis in the study was CompanyX's test process with focus on their test automation.

Research questions: The research objectives were broken down into the following research questions:

RQ1: How was VGT adopted and used at CompanyX for automated GUI-based testing?

RQ2: What are the benefits associated with the short and long-term use of VGT for automated GUI-based testing in practice?

RQ3: What are the challenges associated with short and long-term use of VGT for automated GUI-based testing in practice?

RQ4: What, if any, alternatives are there to VGT for automated GUI-based testing in practice?

Research questions 1-3 aim to support the first research objective whilst research questions 4 aims to support the second objective.

6.3.1 Case company: CompanyX

CompanyX is an application software developer that develops and maintains a product line of applications that share features and backend functionality for streaming music for both desktop and mobile devices. The company's organization consists of 80 loosely coupled development teams located in Gothenburg, Stockholm and New York. Each team is called an autonomous Squad that

consists of maximum ten (10) cross-functional team members [141]. Cross-functionality is needed at CompanyX because each Squad is responsible for the complete development of a feature that is either suggested from management or by the Squads themselves. Each Squad is therefore referred to as its own start-up company that gets to choose what practices, processes and tools they use. A set of standard processes, e.g. Scrum, practices, e.g. Daily Stand-up meeting, and tools, e.g. Sikuli [20], are proposed but teams choose if they wish to use them. The Squads are also grouped into Tribes in related competence areas, for instance the music player, backend functionality, etc. In a Tribe, a squad can still operate with a high degree of independence but is supposed to collaborate with other Squads in the Tribe to implement a certain function or feature, supported by a Tribe lead responsible for ensuring a good environment within the Tribe.

Finally, CompanyX has Chapters and Guilds which are familiarities [141] or interest groups with developers and testers across the different squads. Each developer or tester is associated with a Chapter dependent on his/her skills and competence area within a Tribe. In turn, Guilds are interest groups with specific topics that anyone can create, for instance on the topic of automated GUI-based testing, and anyone interested in the topic can join. A visualization of the organizational structure is shown in Figure 6.1.

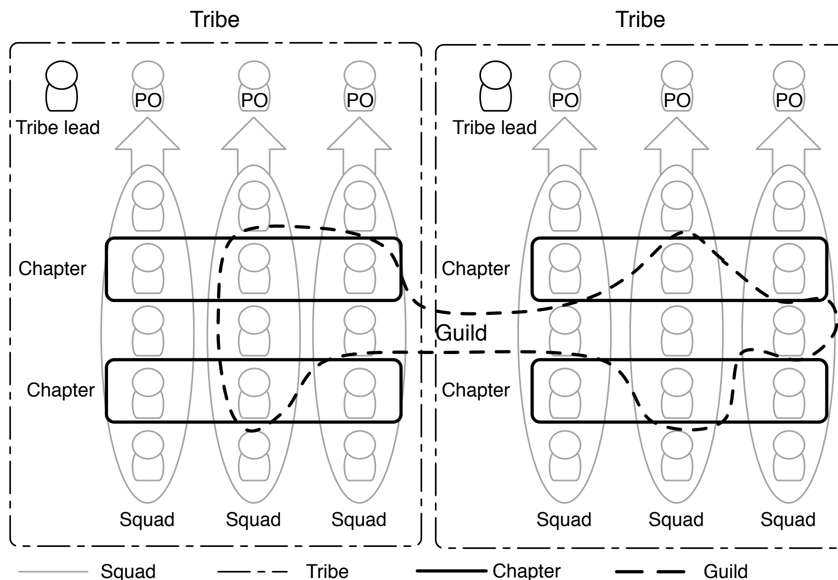

Figure 6.1: *Visualization of the organizational structure used at CompanyX [141].*

The company's core application is divided into a front-end and a back-end where the front-end refers to the GUI and features the user interacts with. Back-end development instead refers to server development and maintenance, i.e. how to stream audio in a scalable manner to the application's millions of users in real-time.

CompanyX's test process includes several automated test activities that are

well integrated into the company's organization. One reason is because the individual teams are responsible for ensuring that each new feature is covered by tests, which endorses collaboration between testers and developers. This collaboration is also required for the company's model-based testing with the open source tool Graphwalker [142] that in some projects requires test code to be embedded in the application's source code.

The automated tests are used for continuous integration but they are not executed on commit, instead the tests are executed automatically from a build server according to a pre-defined schedule. However, no code is allowed to be committed before it has been tested with, for instance, unit, integration or GUI-based tests. These tests are executed on virtual machines to improve test execution time and are designed to be mutually exclusive to enable them to be executed out of order. This practice has been identified as a best practice for VGT in previous work [121] and also allows CompanyX to run subsets of tests to test a specific part of the application.

However, the main purpose of the automated testing is to mitigate repetitive manual testing to allow the manual testers to focus on exploratory testing. CompanyX's test process thereby relies on tight collaboration between developers and different types of testers, supported by a plethora of well integrated test techniques. This process allows the company to release a high-quality application every three weeks on the company's website, Android market or AppStore. However, worth noting is that due to the company's organizational structure, the test practices and test tools differ between different development teams.

Consequently, CompanyX is a highly flexible company with a mixed organization where a multitude of processes and practices are used. The company is therefore representative of both small and medium sized software development companies that use agile processes and practices to develop applications for desktop or mobile devices. They therefore offer a unique opportunity to study the many factors that lead to the successful long-term use of a test automation technique such as VGT.

6.3.2 Research design

The case study was performed in three (3) steps as shown in Figure 6.2.

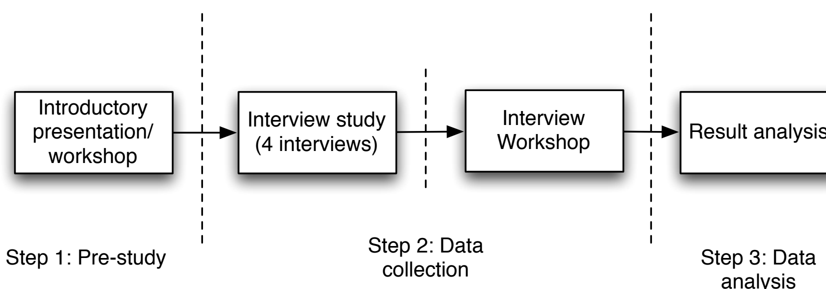

Figure 6.2: *Visualization of the research design.*

Step 1: Pre-study: The first step was a pre-study where an introductory workshop was held at CompanyX to elicit information about the company's

use of VGT, its context, organization, and willingness to participate in the study. The workshop was held at CompanyX's Gothenburg office but was observed over video-link by a group of developers and testers at the office in Stockholm, roughly 50 individuals in total. This workshop began with a 40 minute presentation on testing and VGT, followed by a group discussion (semi-structured interview) with testers, developers and managers at the Gothenburg office. The workshop served to transfer some of the nomenclature that would later be used during interviews and to explain the study's research objectives. Additionally, the workshop participants were used as a seed for snowball sampling [98] to identify suitable interviewees for the study. Hence, interviewees that could provide representative answers regarding how CompanyX works with automated testing, and VGT, to answer the study's research questions. Further snowball sampling was used during each interview to identify the most suitable individuals to interview given the study's limited time and budget constraints. The samplings resulted in six (6) individuals, all proposed by CompanyX, out of which five (5) were interviewed. Triangulation of the sampling results showed that most of the individuals were recommended at all sampling instances that indicate that they were the most knowledgable and suitable people to interview to answer the study's research questions. Further, the sampled individuals had different roles, e.g. testers and test managers[2], and worked in different projects with different platforms, e.g. desktop and iOS. As such, they could provide a representative view of how VGT and GUI test automation is performed at CompanyX.

Step 2: Data collection: The second step of the study served to collect the study's results and was divided into two parts. Part one consisted of semi-structured interviews with the sampled individuals. These interviews gave insight to the success and failure of VGT's use in different contexts, e.g. for different variants of the application developed with different practices and processes, and gave a broad view of what factors that affect the long-term use of VGT in industrial practice.

Four (4) interviews were conducted, where the first interview was held in person with two individuals in a 90 minute session followed by three (3) interviews that were performed over video-link in 60 minute sessions. All interviews followed an interview protocol with 20 questions (See Appendix A) divided into four categories related to the adoption (RQ1-2), use (RQ2-3), maintenance RQ2-3) and abandonment of VGT in some projects at CompanyX (RQ2, 4). The abandonment was studied to identify its cause and to acquire information about the alternative GUI-test approach, i.e. "the Test interface", which replaced VGT. All interviews were recorded and then transcribed prior to analysis.

The interview questions were selected from an initial interview protocol of 36 interview questions that was developed prior to the study. However, due to the interviews' time constraints, the interview protocol was scaled down to 20 questions in a review after the pre-study had been performed.

The second part of step 2 was a 180 minute workshop with one of the lead testers at CompanyX responsible for much of the adoption of current test automation tools and practices at the company. This workshop served to

[2]The ratios between roles, gender, or experience of the employees cannot be disclosed without breaking anonymity agreements with the interviewees.

verify previously gathered results and to acquire information about the current and future use of VGT at the company, in particular how to combine the Test interface with VGT for future use. Hence, the results presented in this paper were extracted from 8 and a half hours (510 minutes) of data elicitation in total.

Step 3: Data analysis: The analysis was performed with a Grounded Theory approach [77, 143] where the qualitative interview results were quantified through open coding [144]. Coding was performed in the TAMSAnalyzer tool [145], which is an open source research tool where the user can define codes and relations to sub-codes. A total of 40 codes were used in the analysis, nine (9) primary codes, presented in Table 6.1, and 31 additional and secondary codes. These codes were defined either before or during the coding procedure to tag specific statements that could support the study's research questions. For instance, to capture statements about CompanyX's test tools, the codes Graphwalker, Sikuli, TestAutomation and TestInterface were defined. The large set of codes enabled deeper analysis if required, e.g. by combining codes to search for specific statements in the TAMSAnalyzer tool, and was required to saturate the interview transcripts with codes [146].

#	Code	Description
1	DevelopmentProcess	Statements related to CompanyX's development process that influenced the testing
2	Graphwalker	Statements about the model-based testing tool Graphwalker that was combined with Sikuli
3	Organization	Statements relating to CompanyX's organization and how it supports their test process
4	Problems	Statements about challenges, problems and limitations with CompanyX's automated GUI-based testing
5	Process	Statements about CompanyX's overall process from requirements engineering to testing
6	Sikuli	Statements about the Sikuli tool, its use, benefits and drawbacks
7	TestAutomation	Statements about the automated testing performed at CompanyX, including processes, tools, etc.
8	TestInterface	Statements about CompanyX's 2nd Generation test tool the "Test Interface"
9	Testprocess	Statements about CompanyX's general test process, including manual and automated practices

Table 6.1: *Summary of the nine primary codes and what types of statements were associated with each code during the interview analysis.*

Coding was performed by going through the interviews and assigning codes to individual statements. A statement could be given more than one code if it was assumed important for several concepts. For instance, the statement;

	Code	Int. 1	Int. 2	Int. 3	Int. 4	**Sum:**
1	Sikuli/Drawbacks	15	10	13	1	**39**
2	Test_interface/Drawbacks	4	4	8	13	**29**
3	Organization	8	2	3	11	**24**
4	Test_interface/Benefits	6	3	4	7	**20**
5	Sikuli/Benefits	4	6	8	1	**19**
6	Sikuli/Adoption	2	5	6	0	**13**
6	Graphwalker	7	4	1	0	**12**
7	Sikuli/Abandonment	2	2	3	0	**7**
8	Testprocess/Manual_testing	2	2	2	0	**6**
	Sum:	**50**	**38**	**48**	**33**	**169**

Table 6.2: Summary of the main tags used during synthesis and the quantities of each tag.

"It (test automation) can of course be more integrated (in the process)... Test automation is still quite new", was tagged with both the "Process" and "TestAutomation" codes. In cases where a larger statement was tagged with one code tag, sub-statements in said statement could be tagged with more specific code tags. Using this approach, 475 code tags were administered in the four transcribed interviews. However, this does not equal 475 unique statements since some statements were tagged with more than one code.

After tagging the interviews, each code was analyzed to synthesize statements connected to individual code tags and draw conclusions. Nine (9) codes were analyzed more rigorously that regarded the use of VGT with Sikuli, manual and automated testing, the organization for the test automation and observed benefits and drawbacks of the different test approaches. These codes have been summarized in Table 6.2 that also shows how many times each code tag was associated with a statement from each interview, and in total, during the analysis. Remaining codes, e.g. "Developmentprocess" were analyzed less rigorously since they did not provide direct support for the study's research questions. However, statements associated with these remaining codes were used to put the main conclusions into context and to define CompanyX's processes, organization, etc. A visualization of the presented coding procedure can be found in Figure 6.3.

The synthesis resulted in 93 low level conclusions from rigorous analysis of 204 statements associated with 169 administered code tags of the analysis nine (9) primary codes. These 204 statements were then grouped to form the main results to answer the study's research questions. As such, each low level conclusion was supported by at least one (1) up to eight (8) statements stored in an excel file to achieve traceability to the main conclusions and preserve the study's chain of evidence [17]. Additionally, several hundred more statements were used to define contextual information to the study's main results.

This analysis procedure was inspired by previous Grounded Theory research based on interviews, for instance Marchenko et al. [140] that used a similar analysis to evaluate the long-term use of TDD at Nokia Siemens.

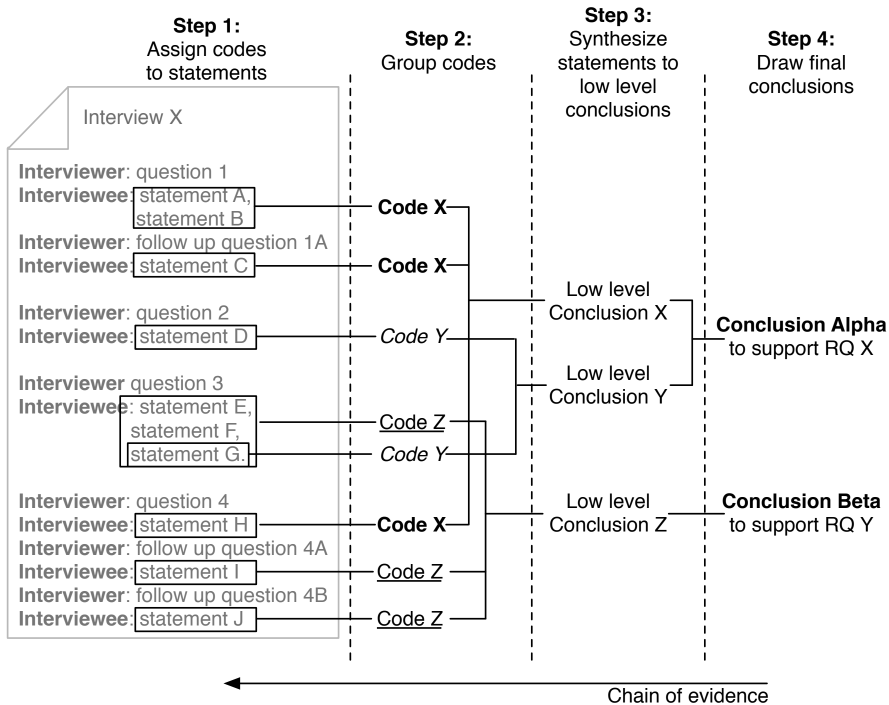

Figure 6.3: Visualization of the coding procedure used during the analysis, where (1) codes were assigned to statements, (2) codes were grouped, (3) groups were synthesized to draw low level conclusions from which (4) final conclusions were drawn. Thus, ensuring a clear chain of evidence from final conclusion to statements given by interviewees.

6.4 Results and Analysis

This section will present the results of the synthesis divided according to the study's research questions. Quotes from the interviews have been added in the text to enrich the results where all quotes are written as: *"Italic text surrounded by quotation marks"*.

6.4.1 Results for RQ1: VGT adoption

VGT, with the open-source tool Sikuli, was adopted at CompanyX in 2011 because of a need for more automated testing of the company's application. Initially the plan had been to add interfaces in the application to support a myriad of different test frameworks. However, due to cost constraints these interfaces could not be achieved. *"We had to create a test interface and knew from the beginning what it should look like. What we did not have, to solve the problem, was resources and possibility to dedicate time to create requirements for the development team (to implement the interfaces)"*. Thus, VGT became the only option for CompanyX since their application lacked the prerequisites required by most other GUI-based test frameworks. The reason why Sikuli was chosen over other available VGT tools was because one of the company's developers had tried it previously and therefore recommended it. *"We were looking for ways to solve the problem (with automated testing), and it was a developer here at CompanyX...that had previously tested Sikuli... that was why it (Sikuli) came to be"*.

The adoption process began with the development of a proof of concept where Sikuli was combined with the model-based testing (MBT) tool Graphwalker [142]. Graphwalker allows the user to create state-based models that can be exported as executable Java programs. Initially all states in the models are empty and therefore require the user to write code that allows the model code to interact with the SUT, e.g. through technical interfaces or by using image recognition technology. These interactions drive transitions between different states in the model that, in this context, represent the GUI state in the SUT. Because Graphwalker models consist of Java code it became natural for CompanyX to adopt Sikuli's Java API in favor of its Python API. *"No, it (the Python API) did not map at all against what we wanted to do, we wanted it in Java"*.

However, the first proof of concept solution was poorly implemented, created by a single developer, as one big script. It was only later that VGT became useful after the script had been broken down into small, reusable, modules through the use of engineering best practices. The use of engineering best practices was one reason for VGT's successful adoption at CompanyX but the main reason was because the adoption was performed by a tight team of engineers that were dedicated to making VGT work. *"A lot of the success relied on engineering solutions and communication because we were developers from both Gothenburg and Stockholm who also worked in different teams with different features. However, we were probably the tightest group at CompanyX because we were adement to develop it (VGT) and continuously make it better"*. Communication during the adoption process served to spread knowledge of best practices and to share reusable components among the adoption team.

Additionally, test scripts were shared and reviewed to ensure their quality.

No explicit changes were made to the tool or the API during the adoption but additional help methods were developed and most of the scripts followed the Page view pattern. *"That we had (Additional methods). We had our own classes with help methods,...I think they were developed straight from the user API"*.

The main challenge during the adoption was to parallelize the test execution and to set up the test environment. *"In the beginning you had to install Sikuli to run it...That was a problem in the beginning when we wanted to set it up on machines for nightly test runs"*. This problem was solved by running Sikuli on virtual machines (VM), but it was still problematic to install everything, Java, Sikuli, etc., on each VM. *"If you have Java on the machine the tests should run. Then it is just important to package our test-jars, the jar that contained the tests, and that it includes all resources that were required (to run the tests)"*. These resources included the Graphwalker Java models, the Sikuli Java API and Sikuli scripts used to interact with the application.

In summary we conclude that for VGT adoption to be successful it requires:

1. An incremental adoption process,
2. Good engineering practices, e.g. patterns and help classes,
3. A dedicated adoption team with good communication, and
4. Virtual environments to run the tests on.

6.4.2 Results for RQ2: VGT benefits

The first observed benefit was the robustness of the Sikuli tool. *"We have actualy not had any stability problems with Sikuli as such, it is actually really good"*. *"Over a whole day, 24 hours, maybe 10.000 pass (Image recognition sweeps) and maybe 8 that fail"*. However, despite the high level of robustness the tool was still reported in two interviews not to be robust enough. *"It depends on the purpose of the tests, if you want to run tests that always go green and pass, then it can be a problem, even if it (failures) happens only then and again"*. *"If you look at it from a positive point of view there was a lot that worked but what failed was in a way annoying enough."*.

The interviewees' perceptions of robustness could however have been affected by the application's high frequency of change, which also required frequent maintenance of the test scripts. For the desktop application this maintenance was considered feasible but for the mobile platforms it was a large issue. *"In our case (Mobile applications) it was not feasible. At one point all I did was to update the images for the Facebook scenario. The desktop application worked better because it was more stabile so there it (the scripts) worked over a longer period of time"*.

However, very few false results had been observed during the use of Sikuli, neither false positives or negatives. False positives were primarily caused by changes to the SUT or due to synchronization problems. *"The main false-positives that we had, they were more in the tests, caused by us not having enough time-outs."*. Further, false negatives were determined to be caused by incomplete scripts rather than challenges with the tool. *"It is possible (That*

a False-negative was reported), but... no, maybe not. Not because of Sikuli, rather because we didn't test it (the defective state)".

Additionally, Sikuli test cases were reported to be reusable between different variants of the application as long as the images used for interaction were updated accordingly. This result implies that maintenance of images can be separated from maintenance of script logic. *"Our fonts are rendered differently between OSX and Windows, we can reuse the tests but we need to change the images".*

The primary reported benefit was however Sikuli's flexibility of use on any platform regardless of implementation which also made it applicable on production ready software. In addition it allowed the testers to incorporate external applications, e.g. Facebook, into the test scenarios which was required since CompanyX supports user login through the user's Facebook account. *"The benefit is that we can use the SUT as a black box. We can use a production grade client, which we have not instrumented or added (test) functionality to. That is... (only) if you can see it you can automate it". "If you want to test things in Facebook, for instance, or kill the app and restart it, ..., then you need to do something outside the app. Then we use Sikuli.".* GUI interaction also made it possible to test the appearance of the GUI, not only its functionality. *"(Sikuli ensures) that you didn't (just) test a button which then turned out to have the wrong color or something like that".*

Sikuli's Java API was also reported as a benefit since it made it possible to, in a flexible way, code additional functionality into the test cases or the test framework when required. As such, workarounds could be created when conventional use of the API was not enough. *"What (test functioanlity) is missing we can simply code. You can find workarounds for most things in Sikuli, but it (the test) becomes more complex".*

Another benefit with the Java API was that it integrated well with Graphwalker [142] for MBT based VGT. *"Then (for Graphwalker) Sikuli fit well since it provides a Java API...It fit like a hand in a glove so there were absolutely no problems".* Graphwalker's will be described in more detail in Section 6.4.4.

Sikuli was also reported as a useful and valuable tool for finding system regression defects, especially during periods when CompanyX's client has been unstable. However the client instability had also resulted in additional maintenance of the test scripts that was not considered feasible at that time. *"Yes, we did (find defects), primarily because our client broke continuously. So the defects we found were often that the client crashed when you entered the artist-view, or similar. In that way it was a positive thing, even if it felt as, or actually was, unfeasible to maintain, it still contributed to the defects in the system being found".*

In summary we conclude that the benefits with Sikuli, and VGT, are that:

1. Test scripts are robust both in terms of execution and number of false test results,

2. Test script maintenance is considered feasible for desktop applications,

3. Test script logic can be reused between different variants of an application,

4. Test scripts are flexible and can be used to test the actual product as well as incorporate external applications with limited access to the test cases,

5. Sikuli integrates well with Graphwalker for MBT based VGT, and

6. Test scripts find regression defects with equal ability as manual system tests.

6.4.3 Results for RQ3: VGT challenges

The main drawback with Sikuli reported by CompanyX is its limited use for GUIs that present dynamic data, i.e. non-deterministic data from, for instance, a database. Whilst all VGT tools can verify that non-deterministic data is rendered by checking that a GUI transition has occurred, the tools require a specific expected output image to assert if what is being rendered is correct. In CompanyX's case this presented a problem since much of the application consists of dynamically rendered lists of songs, artists and albums. Whilst tests could be performed on a stable database, with specific search terms, the company wanted to run the tests in the real production environment where a search for a set of songs does not always return the same list. Thereby impeding Sikuli's usefulness. *"The test data we have includes a lot of songs, albums and artists and such. They have different names, cover arts... it is very hard to verify that it is the correct image for each artists name". "It is difficult to work with them (tests) in Sikuli, they need to scroll (in lists) and it is difficult to distinguish different rows, they look the same. Big buttons are very easy."*.

Attempts to solve this problem included using Sikuli's optical character recognition algorithm (OCR) and to copy the entire list to the clipboard and then importing the clipboard to the scripts for analysis. However, both solutions were found unreliable. Sikuli's OCR has until recently had a very low hit rate and copying text often failed because the key commands did not work properly in the application. *"We have verified which songs are in a playlist. We selected all songs, copied them to the clipboard but sometimes this process failed...ctrl A and ctrl C did not work. This is a fault that is not relevant for CompanyX". "Then I need to extract that information (song list) dynamically somehow. You could use the OCR functionality in Sikuli, but it is way too unstable"*.

Another large drawback was the amount of image maintenance that was required. CompanyX's application is developed for a myriad of different platforms that all render the GUI slightly differently and have different, operating system specific, images. Thus, each time the GUI was changed, the images for that change had to be maintained in each test suite for each variant of the application. Further, since CompanyX interacts with external applications, like Facebook, the test suites also had to be maintained for changes made to software outside CompanyX's control, i.e. maintenance that could not be foreseen. *"It could be both (images and test logic), but it was probably most often the images.". "Even if we remove everything that has to do with Facebook and what is outside our control, even if we would use Sikuli entirely for our own app, we would have problems (with image maintenace)". "But the problem*

with Facebook was that they change as much as we do. The difference is that we have no idea when and what they change.". As such, much of the image maintenance problem came from automation of external software applications, which was also where Sikuli was the most beneficial since there was no other way to access and/or stimulate those applications automatically.

Yet another drawback was that Sikuli was experienced to have limited applicability to test applications on mobile devices. However, several of the interviewees stated that they did not know the current status of the mobile VGT, which, for instance, is support by the VGT tool EggPlant [68]. *"On mobile phones, if you want to run on an actual device, it doesn't work. I havn't check the last year, maybe they have worked on adding such support?".*

Further, when Sikuli scripts execute, they take over the mouse and keyboard from the user. This implies that the user cannot use the computer at the same time as a script is running. *"It is according to me a problem that has to be solved. It is way to ineffective otherwise. If you write a script that takes fifteen minutes to run, then you don't want to lock up the computer for fifteen minutes, you want to debug other tests at the same time".* Additionally, Sikuli scripts were perceived to execute slowly, especially for larger test scenarios. *"Yes, I guess it is a problem (Slow test execution). On the other hand the tests are system tests, so the actual problem is maybe not Sikuli's fault".* This problem was assumed to be solvable by distributing the test execution over several virual machines (VMs) and/or reference systems with physical devices, i.e. parallel test execution. However, CompanyX had experienced problems with running the test cases this way because they would not execute if there was no physical screen connected to the computer. This problem originates in Sikuli's use of the AWT Robot framework that only initiates if a physical screen is connected. *"In our test lab we don't have monitors for all machines that we have and there is something strange with that. The desktop tests worked fine but for the mobile application tests something strange happened if it (Sikuli) did not detect a screen, then it was not possible to run Sikuli-stuff.".*

In summary we conclude that the challenges with Sikuli, and VGT, are that:

1. Test scripts have limited use for applications with dynamic/non-deterministic output,
2. Test scripts require significant amounts of image maintenance,
3. Sikuli scripts have limited applicability for mobile applications at CompanyX, and
4. Sikuli locks up the user's computer during test execution.

These challenges have resulted in CompanyX's abandonment of Sikuli in many projects in favor of a 2nd generation test approach that will be discussed in Section 6.4.4. The main reasons for the abandonment were the high costs associated with maintaining images and the tool's lacking applicability for mobile applications. These challenges were as such the main long-term, continuous, challenges that CompanyX experienced with VGT. Thus, answering research question 3. However, it should be noted that, at the time of writing this report, Sikuli is still used at CompanyX for testing of the Desktop and

Web applications in combination with other automated and manual test approaches. One reason is because these projects require less variants of the test suite that also mitigates the amount of image maintenance after change. In contrast, the mobile applications required individual test suites for different devices, screen resolutions, etc.

Results from the the workshop in phase 3 also indicate that the adoption of VGT at CompanyX was instrumental for the current test automation culture at CompanyX. Hence, when developers saw the benefits of test automation with VGT, several developers took it upon themselves to do the necessary refactoring required to make additional automation techniques applicable.

6.4.4 Results for RQ4: VGT alternatives

This section provides an overview of two of the main tools that CompanyX use for automated GUI-based testing.

Graphwalker: As mentioned, CompanyX's automated GUI-based testing is based on a model-based testing framework called Graphwalker. Graphwalker models are created graphically in a tool called Wired and then exported to a Graph ML format. These models can then be used to create executable Java class stubs, i.e. empty methods, in which the user defines the interactions with the SUT.

Each model defines a linear test scenario but several models can be linked together through a meta-model to create more complex test scenarios. This ability also allows the user to reuse scenarios, e.g. a login scenario, to lower development costs. *"If you have general scenarios (that run) in several views, you can make that scenario into a model and simply switch to it every time.. you get an overview model and you can reuse the scenario in other scenarios. It (Graphwalker) also has support for conditions (for branching scenarios), which are usually states. For instance if we have a login scenario, then we set that if Login=True then it knows that state in the model and may not go to another state before the condition is True (in the application)"*. Thus providing some parameter based programming support in the models themselves.

In addition, Graphwalker supports random, real-time, traversal of models for automated random testing, with traversal algorithms such as A* [147] and random. *"Yes, you get many permutations of the model and a lot of interesting things happen when you run the tests this way (automated traversal). However, we traverse our models with so called on-line generation, which means that we don't generate a path from the model that we then use. Instead we always ask Graphwalker to generate the path in run-time. Mostly we used the random generator, which gives us different permutations. What we want to ensure is that we have full coverage (node coverage) of the model. So, the generators are what makes it possible for us to traverse the model in different ways. Stop conditions make it possible to express when we are done. We don't have any stop nodes in the model"*.

Test interface: As mentioned in Section 6.4.1, CompanyX's intention was originally to add test interfaces to the application for automated testing. However, the test interfaces had not been achievable due to resource constrains. When Sikuli was abandoned, the need for automated GUI-based testing once again presented itself but due to the cultural changes in the company, with a

greater focus on automated testing, CompanyX's initial plan could be realized.

The solution, simply called the *Test interface*, is a 2nd generation GUI based testing approach where hooks (test methods), are embedded into the source code. These methods are designed ad hoc to provide the tester with the state information (s)he needs to write a test. The Test interface is accessed by test case scenarios that are defined in Graphwalker, some reused from previous Sikuli testing.

Benefits: Several benefits, but also drawbacks, were reported with the Test interface solution. The primary benefit is the flexibility the Test interface provides CompanyX to perform, for instance, tests with dynamic data, e.g. playlists, and other test objectives not fully supported by Sikuli. *"There is support for anything really, but it requires that you write it (the test support) yourself. Hence, if you, for instance, want swipes then you have to add a method in the interface that actually does that...if you have a tabel, a playlist, where there are songs, then there is for each table, playlist, methods to scroll to an item at an index... you don't do that on the real UI layer"*.

Further, the execution speed and stability of the Test interface is perceived higher than for the Sikuli tests. *"(The benefits are) Time, it is faster with the new (Test interface). Stability (Robustness)"*. In addition, the execution time can be improved since the Test interface can modify how the application behaves, e.g. it can remove animations between state transitions. *"The difference (to Sikuli) is that with the Test interface we can remove animations, so when you open "Playing view", the player, then we set the test interface not to animate but instead render (the output) instantly"*. Further, the Test interface runs in the background without locking up the computer, unlike Sikuli, which allows the user to continue work whilst the tests are running. This approach is also more stable since it allows extraction of different types of data directly from the application, which make interactions and assertions more exact. *"The benefit is that you can read the unique identifiers that each song has...I can go to an album and read which songs are there and save them. Then I can go to "Add to your music". Then I can go to "Your music" and assert if they are there (The unique identifiers)"*. Thus solving the dynamic data assertion problem that CompanyX experienced with Sikuli, presented in Section 6.4.3 whilst also improving the tests' robustness. *"I would blindly ship this (The application) to employees being sure that 90 percent of the application would work"*.

However, the Test interface flexibility and robustness comes at the expense of opening the application up to explicit internal access to its functions and features, which also presents a threat of user misuse. *"We have full insight into the client code...this Test interface is very open, you open up the client to do what you want"*. This threat is removed by a test architecture, abstract model shown in Figure 6.4, where each loosely coupled Test interface method is coordinated by an orchestrator class within the application. Before the product is released, the orchestrator class is removed from the application which turns all the test interface methods into "dead code", i.e. code that is unreachable in the application. A tool called ProGuard is then applied which removes all dead code, effectively removing all the test interfaces. *"ProGuard is a tool in which... will remove those endpoints (methods) because they are not in use anymore."*.

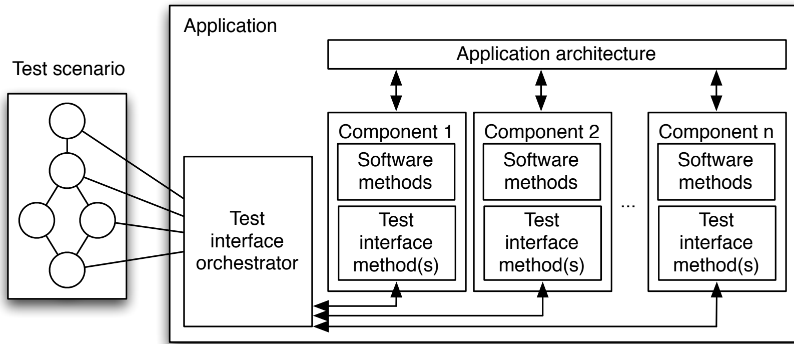

Figure 6.4: *Visualization of the Test interface architecture within CompanyX. All Test interface methods are accessed through a test orchestrator, which connects test case nodes to specific Test interface methods.*

Another benefit of this architecture is that if changes are made to the application's source code that breaks the coupling to a Test interface method(s) the developer will receive a compilation error. As such, the developer gets an instant notice when, and what, Test interface methods requires maintenance. *"The plan is that the Test interface is part of the code such that if you change a feature you should get a compilation error... and you never get unstable tests."*.

The transition to the Test interface has required a huge investment and organizational change but is assumed to have lowered test maintenance costs compared to Sikuli. *"Yes, I would claim that (Perceived lower maintenance costs). However, it is difficult to say because we have also improved our process. We have hired people that are dedicated to each platform, before we (Small test team) had to do these parts (Test maintenance)"*. Thus, the lowered maintenance costs are caused by a combination of factors but the previous image maintenance costs have been removed since images are no longer used for interaction and assertion of the applications correctness.

Drawbacks: However, this leads to the test interface first drawback, it does not verify that the rendered, pictorial, GUI is correct. *"We can miss bugs now, for instance... we do not notice if the client is upside down (GUI rendered incorrectly)"*. *"What we miss now when we run the Test interface is the UI part. We see if the functionality works but we don't know if it (the GUI) looks right. We could see that with Sikuli, at least partially"*.

Further, interactions with the application during testing is not performed in the same way as a user interacts with the software. Hence, instead of clicking on components, these interactions are invoked from layers underneath the pictorial GUI. *"But when we build our own interfaces, then we are clicking, in a way, from beneath"*. *"We create events that essentially do the same thing but without the physical click"*. Further, because the test interface code is removed in the release ready version of CompanyX, the tested version of the application is not the same as what is delivered to customers. *"The main drawback, which we knew from the beginning, is that we're not testing the real products, we're testing something else, more or less"*.

Synchronization between the test cases and the SUT was presented as a challenge with Sikuli. However, the same challenge has been observed with the Test interface. *"In that regard it (Synchronization) is the same. It looks reasonably the same independent of if it is Sikuli or the Test interface. You have to solve it in different ways, but the core problem is the same. One part of the challenge with test automation is how to deal with asynchronous test execution"*.

Another common problem for both Sikuli and the Test interface is that none of them actually verifies that CompanyX plays music, i.e. auditorial output. *"We have manual testers that go through stuff (e.g. that music is playing), so we capture those things. It is not a big risk that a version of the application reaches the customer without sound"*.

In summary we conclude that CompanyX use Test interfaces embedded in the source code, driven by a model based testing solution, i.e. Graphwalker, for GUI-based testing with the benefits that:

1. The Test interface provides more flexibility of use to test specific parts of the CompanyX application, e.g. lists, than Sikuli,

2. Test interface tests execute quicker and more robustly than Sikuli tests, and

3. Broken test cases are instantly identified by coupling to the software components that generate a compilation error if a test interface method requires maintenance, ensuring that they are continuously up to date.

However, the Test interface still has drawbacks, for instance that:

1. Test interface test cases do not verify that the pictorial GUI is correct, only the functionality,

2. Test interface interaction with the application differs from human interaction, i.e. interactions are invoked rather than performed through the user's means of interaction,

3. Synchronization between Test interface test cases and the application is still a challenge, similar to Sikuli, and

4. Neither Sikuli or the Test interface are able to verify that the application actually plays music.

6.4.5 Quantification of the Qualitative Results

The study reported many benefits and challenges with both Sikuli and the Test interface. To provide an overview of the qualitative results they were quantified based on a set of properties observed for both techniques. Quantification was done on a five (5) point scale from low (1) to high (5) as shown in Table 6.3. The table implies that, for instance for the robustness property, VGT was considered less robust than the Test interface.

The quantification was made based on the amount of support for and against each property for both techniques, taking all interview and workshop results into account. We do however stress that this quantification is based on

Property	**VGT**	**Test interface**
Ease of Graphwalker integration	5	5
Robustness	3	5
Frequency of correct test results (No false positives or negatives)	4	4
Defect finding ability	4	3
Migratability of tests between SUT variants	3	3
Feasibility/Maintainability of scripts	2	5
Support for Parallel test execution	3	4
Ease of Synchronization between tests and SUT	3	4
Speed	2	5
Flexibility of integration with different plattforms	4	4
Desktop	5	5
Android	3	4
iOS	3	3
Web	5	3
Flexibility of use for different testing	3	4
Dynamic data	2	5
Work in parallell to test execution	2	5
Test of external software	5	1
Test o auditory output	1	1
MBT support	5	5
Support for different types of tests	5	3
Acceptance test	5	3
System test	4	5
GUI-based testing	5	1

Table 6.3: *Quantification of the interviewees' perceptions of the VGT solution compared to the Test Interface. Each property is ranked on a scale from 1 to 5 where 1 is low and 5 is high. Example: A Robustness of 5 implies high Robustness.*

data from CompanyX's context and are only estimates based on the qualitative results. The quantification was however performed by an expert that took previous work and experience with both 2nd and 3rd generation GUI-based testing into account.

The quantified results were then analyzed statistically with the non-parametric Mann-Whitney U test to test if there was any statistical significant difference between the two techniques. A non-parametric test was used since normality analysis with the Shapiro-Wilks normality test showed that the samples were not normally distributed, with p-values 0.0069 and 0.00039 for the VGT sample and Test interface sample respectively. The result of the comparison showed that we can not reject the null hypothesis, p-value of 0.473, and therefore we conclude that there is no statistical significant difference between the two techniques in terms of their properties. However, analysis of the statistical power of the two samples showed that it was only 0.109. Hence, well below 0.8 which implies a chance of Type II error, i.e. that we failed to reject the null hypothesis despite it being false. These results are further discussed in Section 6.6.

6.5 Guidelines for adoption and use of VGT in industrial practice

In this section we present a set of practitioner oriented guidelines for the adoption, use and long-term use of VGT. These guidelines, summarized in Table 6.4, were created through qualitative synthesis and triangulation of solutions, guidelines, factors, etc., for best practice VGT and automated GUI-based testing presented in previous [50, 121] and related work [99]. The purpose of the guidelines is to provide practitioners with decision making support as well as guidance to avoid common pitfalls with VGT. However, it should be noted that future work is required to expand this set of guidelines and verify their validity and impact in other companies and domains.

Phase	#	Guideline	Description	Support
Adoption	**1**	**Manage exectations**	It is not suitable/possible to automate anything and everything with VGT, consider what is automated and why?	A, B, C, D
	2	**Incremental adoption**	A staged adoption process that incrementally evaluates the value of VGT is suitable to minimize cost if the technique is found unsuitable.	A, B, C
	3	**Dedicated team**	Dedicated teams do not give up easily and identify how/when to use VGT.	A, B, C
	4	**Good engineering**	VGT costs depend on the architecture of tests/test suites and engineering best practices should therefore be used, e.g. modularization.	A, B, C, D
	5	**Software**	Different software solutions, e.g. VGT tools and third party software, should be evaluated to find the best solution for the company's needs.	B, C, D
Use	**6**	**Roles**	VGT requires training of new roles, which is associated with additional cost.	A, B
	7	**Development process**	VGT should be integrated into the development process, e.g. definition of done, and the SUT's build process, i.e. automatic execution.	A
	8	**Organization**	Organizational change disrupts development until new ways of working settle.	B
	9	**Code conventions**	Code conventions improve script readability and maintainability.	B
	10	**Remote test execution**	For distributed systems, VGT scripts should be run locally or use VGT tools with built in remote test execution support	B, C
Long-term	**11**	**Frequent maintenance**	The test process needs to prevent test cases degradation to keep VGT maintenance costs feasible long-term.	D
	12	**Measure**	The costs and value of VGT should be measured to identify improvement possibilities, e.g. new ways of writing scripts.	D
	13	**Version control**	When the number of SUT variants grow, so do the test suites and they should therefore be version controlled to ensure SUT compatibility.	D
	14	**Life-cycle**	Positive return on investment of VGT adoption occurs after at least one iteration, so how long will the SUT live?	B, C

Table 6.4: *Summary of guidelines to consider during the adoption, use or long-term use of VGT in industrial practice. Column "Support" shows if this study (A) or which previous work, B [121] and C [50] and related work D [99] that supports the presented guideline.*

6.5.1 Adoption of VGT in practice

Manage expectations: VGT is associated with high learnability and ease-of-use that makes it tempting to use it for automation of all types of test cases. However, VGT is primarily a regression test technique for system and acceptance tests and is therefore not suitable for testing of immature or frequently changing functionality in the SUT since the maintenance costs of such scripts will be high. Test cases that are developed in early stages of VGT adoption should therefore be removed after exploring what types of SUT functionality they can test.

Another common expectation is that VGT can completely replace manual testing in an organization but this is not the case since VGT scripts can only find defects in system states that are explicitly asserted. In contrast, a human can observe faulty SUT behavior regardless of where or how it manifests on the SUT's GUI and VGT scripts therefore need to be complemented with manual test practices, e.g. exploratory testing [116].

Use incremental adoption: Large scale adoption is seldom recommended for any new technique or practice, and the same applies to VGT. Instead, VGT should be adopted in a staged/incremental adoption process with one or several pilot projects to evaluate the technique with several different VGT tools. The reason is because these tools have different capabilities that make them more or less suitable based on contextual factors such as the test automation culture of the company, if the system is distributed, if the testers have programming knowledge, etc. Additionally, the pilot projects should strive to find suitable test cases to automate and take maintenance costs into consideration. Hence, these projects need to span over a longer period of time, at least a few months, to evaluate as many aspects of the development and maintenance of scripts as possible. Thereby ensuring that a correct decision can be taken for the technique's full scale adoption or abandonment.

Use a dedicated team: VGT is easy to use but it is associated with many challenges that can stifle a team's progress and lead to developer frustration. A team of dedicated individuals should therefore drive the adoption process such that the technique is not abandoned prematurely, i.e. before all aspects of the technique have been evaluated.

Use good engineering: VGT scripts, especially in the open source tool Sikuli [20], require a deal of engineering to be as usable and maintainable as possible. For instance, VGT scripts should not be adopted as 1-to-1 mappings of manual test cases if these test cases include loops or branches since this will make the scripts more difficult to read and maintain. Instead, test cases of this type should be broken down into as short and linear test scripts as possible. These scripts should also be written in a modular way where script logic is separated from images to make the logic and images reusable between all test scripts in the test suite [99]. This practice supports maintenance since changes to the SUT only require script logic, or images, to be maintained in one place for the entire suite. Further, VGT scripts must be synchronized with the SUT's execution, synchronization that should be added systematically to the scripts, preferably in a way that makes it possible for the user to change script timing globally for the entire test suite simultaneously. Additionally, it is a good practice to add failure mitigating code in the scripts, for instance by having

assertions rerun if they fail, to ensure script robustness. However, care should be taken with this practice since emphasis on robustness has negative effects on the scripts' readability and execution time. Finally, VGT scripts should be documented to improve readability and to ensure that reusable script modules are easily accessible.

Consider used software: An automated test environment often consists of more than the tool and the SUT, it also contains simulators, third party software, build systems, etc. Different VGT tools have different built in capabilities to integrate with this environment that further stresses the need to evaluate different VGT tools during a pilot project. Additionally, if environmental software components are interchangeable, e.g. remote desktop or VM clients, several options should be tested to find the best possible solution for the company's context.

6.5.2 Use of VGT in practice

Change roles: Adoption of a new technique can require new roles to be formulated, e.g. a role dedicate to the development and maintenance of scripts. However, an individual placed in such a role needs training and/or time to familiarize themselves with the new technique which also adds to the adoption costs of the technique. Additionally, the new role's responsibilities will change that must bee accounted for when planning the individual's workload.

Consider the development process: VGT should be integrated into the development process to be effective. This implies adding the technique to the normal routines at the company for instance by adding VGT to the definition of done (If such is available) of a feature or task. Additionally, the scripts should be added to the company's build and test process to allow them to be executed automatically and frequently, e.g. every night. Further, it has been found that changing the order of the test scripts between test executions can have a positive effect on the scripts' failure-finding ability [121].

In addition, the company needs to consider for what purpose VGT is used and cover other test related needs with other techniques. VGT is primarily a regression test technique but it can also be used to provide stimuli to a SUT during long-term tests to make these tests more representative of use of the SUT in practice. Hence, it is perceived that VGT can be used for more than regression testing, but, as stated, it cannot replace manual testing. Therefore, the adopting company needs to redefine their test process to make use of all the company's test techniques' benefits in the best way possible.

Change the organization: With changes to roles and the development process comes also changes to the company's organization, e.g. diversion of responsibilities between individuals, new co-workers, etc. These changes can disrupt development for a time, which will have monetary impact before the new processes and organization settles. The impact of this organizational change can vary dependent on how VGT is adopted but it is suggested that VGT knowledge is spread across the organization but primarily handled by dedicated individuals, as reported in this study and previous work [121].

Define code conventions: Code conventions keep scripts consistent that make them easier to read and maintain. Additionally, these conventions can be used to convey how specific VGT related practices, e.g. systematic syn-

chronization, should be performed by the developers or how the code should be structured to promote modularization, reuse and maintainability.

Minimize remote test execution: VGT tools can be executed on top of remote desktop or VNC clients to facilitate testing of distributed systems. However, results from previous work [121] indicate that this practice has adverse effects on some VGT tools' image recognition success-rate. Therefore, for distributed systems, it is recommended that the company uses a VGT tool with built in VNC functionality, e.g. EggPlant [68], to mitigate these adverse effects. Another practice, recommended in previous work, is to minimize the use of VNC and run the scripts locally to the greatest extent possible. This practice implies that certain test cases become out of scope for VGT, the impact of which should be evaluated during the pilot study.

6.5.3 Long-term use of VGT in practice

Adopt frequent maintenance: To avoid test script degradation it is important to frequently maintain and improve the test scripts [99]. Frequent maintenance also helps lower test maintenance costs since it avoids, to a larger extent, simultaneous maintenance of both logic and images that is more complex than logic or images separately. As such, a maintenance process should be integrated into the company's overall development process to ensure that changes to the SUT have not caused scripts to break. Dependent on how VGT has been adopted, this maintenance process either requires common knowledge among all developers of how and when to update the scripts or clear communication channels to the individual(s) responsible for test script maintenance. Regardless, frequent maintenance is an important activity to ensure the long-term feasibility of VGT scripts in practice.

Measure for improvement: Measuring the status of the VGT process is important to gauge its value contra the costs of performing it, for two main reasons. First, to evaluate if VGT is beneficial for the evolving SUT, i.e. is the technique equally suited to test new features of SUT as it was when the technique was adopted? For instance, is it suitable to test the new features through the pictorial GUI or is a lower level automated test technique more suitable? Second, VGT scripts are large and slow in comparison to many other lower level test techniques. This implies that a VGT test suite becomes saturated quickly if a dedicated time slot is allocated for the test suite's execution. Especially since VGT scripts execute in the order of minutes and test suites in the order of hours. Hence, if VGT is used for continuous integration it may quickly become necessary to do test prioritization and test suite pruning [99], which is non-trivial without proper measures to identify what test cases to change, remove or execute for a specific test objective. Such a measurement scheme should therefore be put in place as soon as possible after the technique's adoption.

Version control scripts: As reported from CompanyX, it is possible to reuse test script logic between variants of a SUT but not the images. However, the variants of CompanyX's applications share a lot of functionality which is not generally the case in practice. Further, the features and functionalities of variants of a SUT can diverge over time, which implies that the script logic cannot be reused. However, in some cases it may be required to migrate or

reset old test cases to an old variant of the SUT, which implies that VGT scripts should be version controlled together with the SUT's source code to ensure script compatibility with different variants and versions of the SUT.

Consider SUT life-cycle: VGT scripts are associated with a development cost that requires the scripts to be executed several times before they provide positive return on investment. Further, the test scripts are used for system and acceptance testing that implies that they can not be created before the SUT has reached a certain level of maturity. As such, they need to be formulated later in the development cycle and are therefore better suited for SUTs that will go through more than one development iteration or be maintained for a longer time period. Hence, for small projects where the product will be discontinued after the project, e.g. development of a one-off or a prototype, it may not be feasible, even suitable, to adopt VGT. Instead, manual regression and exploratory testing should be used.

6.6 Discussion

The main implication of the results presented in this work is that VGT can be used long-term in industrial practice. However, care must be taken how the technique is adopted and used for it to be feasible and to mitigate its challenges. This implication stresses the need for best practice guidelines regarding the adoption and use of VGT in practice, i.e. guidelines such as those presented in Section 6.5. The guidelines presented in this work are however not considered comprehensive and further work is therefore required to expand this set and evaluate their usefulness and impact in different companies and domains. As input to such research, best practices for traditional software development could be analyzed and migrated for use in VGT scripting.

However, despite following such best practices, the challenges associated with VGT proved too much for the technique's continued use in several projects at CompanyX. Primarily this was due to high maintenance costs and because of Sikuli's limited ability to test mobile applications, i.e. run tests in the mobile device. It is possible that these challenges could have been mitigated by, for instance, pruning the test suites to focus only on stable SUT functionality and GUI elements, a statement supported by the result that the technique was feasibly used for CompanyX's desktop and web applications. Additional mitigation could have been achieved by adopting another VGT tool, e.g. Eggplant [68], which has better support for mobile testing. However these challenges can not be ignored and they, image maintenance in particular, should therefore be studied further to find means of mitigation, both through process improvements and technical support.

Further, Section 6.4.5 presented a comparison between Sikuli and CompanyX's Test interface solution that showed no significant difference between the techniques. For instance, both techniques were found equally easy to integrate with the Graphwalker MBT framework that improved both techniques' applicability and feasibility, e.g. by supporting migration of scripts between applications, improved maintainability of scripts, etc. The presented analysis did however not cover the costs of the techniques' adoption, where the Test interface approach was considered significantly more costly than VGT, which

is also the reason why CompanyX adopted Sikuli in the first place rather than the envisioned Test interface solution. However, the most significant difference between the approaches was the maintenance costs, where maintenance of images for the Sikuli scripts were considered significantly more costly than maintenance of the Test interfaces in some projects. Further, the Test interface could be tailored to fulfill test objectives not supported by Sikuli, for instance testing of non-deterministic outputs. However, the Test interface had its own limitations, the primary being that it, common to all 2nd generation GUI-based tools, lacks the ability to emulate user behavior, i.e. stimulation and assertion of the SUT through the same interfaces as the human user. Instead, the Test interface invokes interactions from beneath the GUI that is suitable to test the application's functionality but does not verify if a user can access this functionality or that the SUT's GUI appearance is correct. This implies that the Test interface requires more complementary manual testing than VGT, which also limits the technique's use for continuous delivery where a new feature should be built, verified and validated, shipped and installed at the customer automatically on each commit [45].

As such there are several tradeoffs between the properties of the two techniques in terms of speed, robustness, flexibility, cost, etc. However, many of these properties are complementary, which implies that a combination of both techniques could provide additional benefits, a sentiment also shared by CompanyX. *"That would get rid of those parts (test tasks cumbersome in Sikuli)... a combination with Sikuli and this (the Test interface) would then be a solution"*. This sentiment also supports the conclusion drawn in previous work that, in an experimental setting, showed the value of combining VGT testing with 2nd generation GUI-based testing to mitigate false test results [51]. This previous work also evaluated automated model-based GUI-testing for test case generation and execution, work that is also supported by this study since CompanyX's Graphwalker solution supports automated random testing of the GUI. Hence, technology that theoretically could be advanced to support automated exploratory GUI-based testing to further mitigate the need of manual testing of software applications. Thus, another topic that warrants future research.

Another interesting observation from this study was that the Test interface fruition was caused by the adoption of VGT that changed the automation culture at CompanyX. The implication of this observation is that VGT could be a suitable first step for a company to improve their test automation practices. Especially in contexts where software legacy or lack of test interfaces prevent the use of other test automation frameworks.

Consequently, this study provides valuable insights regarding the long-term perspective of using automated GUI-based testing in industrial practice, including challenges for the long-term use of these techniques. Based on these results it was possible to synthesize practitioner oriented guidelines, which shows the value and need for this type of research This study thereby provides a concrete contribution to the limited body of knowledge on VGT but also a general contribution to the body of knowledge on software engineering that currently includes very few studies that focus on the long-term perspective of industrially adopted research solutions [138, 140]. Hence, studies that are required to draw conclusions regarding the impediments of such research solutions and improve their efficacy, efficiency and longevity in industrial practice.

6.6.1 Threats to Validity

Internal validity: Several measures were taken to ensure the internal validity of the study's results. First, the interviewees were carefully chosen through snowball sampling based on expert knowledge from within the case company's organization to ensure that the sample could provide representative results to answer the study's research questions. The study's time and budget constraints also made it possible to interview all but one of the interview candidates, which were spread across the organization and different projects. Additionally, the interviews were triangulated with results from two workshops that were attended by experts and individuals from all over CompanyX's organization and provided results that also supported the interviews.

Second, to ensure a clear chain of evidence, the interviews were fully transcribed, coded and analyzed with a Grounded Theory approach [77] that was modeled on previous research on the long-term use of TDD [140]. This allowed low level conclusions to be drawn that were supported by one (1) to eight (8) statements each. These low level conclusions were then grouped further to draw the study's final conclusions, thereby ensuring that each conclusion was well grounded in the representative interview sample's statements.

Third, the guidelines presented in this work were triangulated with third degree data from both previous and related work. These guidelines were thereby supported both by the rigorous analysis in this study but also triangulated with external sources of evidence. The use of related work also helps mitigate bias in the guidelines fruition.

External validity: Only one case company was used for this study, chosen through convenience sampling, since there are few companies available that have used VGT for a longer period of time, i.e. companies that meet the prerequisites of this type of study. However, CompanyX's organization is based on self organizing teams, called Squads, which are treated as their own startup companies that choose their own processes, practices and tools. Squads, which consist of maximum 10 individuals, are also grouped into feature teams, called Tribes, which consist of several Squads. The interviewees chosen in the study came from different Squads and Tribes that raises the external validity of study's results to both small and medium sized companies where a medium sized company would be in excess of 50 employees but less than 100 employees.

Further, the VGT guidelines presented in this work were triangulated with third degree data that was acquired in other companies and domains, including larger safety-critical software development companies. Thereby ensuring that the guidelines are generalizable for both small agile companies as well as large safety-critical software developers. Additionally, the related work was based on research with other types of automated GUI-based testing, which implies that the reported guidelines can be generalized beyond VGT.

However, future research is required to build on this work and expand it to other companies and domains to verify its results. This study is therefore fundamental for such research since the reported results can help more companies avoid the pitfalls associated with automated GUI-based testing and only then reach a state of maturity where this type of research can be performed.

Construct validity: The study was performed with interviews and workshops with industrial experts with years of experience and knowledge about

VGT and automated testing in industrial practice. One interviewee was also part of the team that adopted VGT at CompanyX. As such, these subjects could provide in depth answers regarding VGT's life-cycle in their context and valid support to answer the study's research questions.

Further, the guidelines presented in this work were primarily triangulated with empirical research on VGT performed in industrial practice, thereby ensuring the guidelines validity to help practitioners.

Conclusion validity/Reliability: To improve the reliability of this study, as much detail as possible has been presented regarding the case company, the research process as well as how analysis was performed. These measures should make it possible to judge the validity of this work as well as to replicate the study in similar contexts to the one described. Additionally, references have been added to clarify the methods used in this work, e.g. case study design [17, 71], Grounded theory [77], open coding [144], snowball sampling [98], etc., to endorse the study's replication.

6.7 Conclusions

This paper presents the results of a single, embedded, case study focused on the long-term use of Visual GUI Testing and automated GUI-based testing at the software application development company CompanyX. The case study's results were acquired from two workshops and four (4) interviews with five (5) carefully chosen individuals from different parts of CompanyX's organization to provide a representative view of the company's use of automated GUI-based testing. These results were then analyzed using Grounded theory to acquire the study's main conclusions.

Based on the study's results it was concluded that VGT, with Sikuli, can be used long-term in industrial practice but that there are many challenges associated with the technique, e.g. high maintenance costs of images, that some VGT tools have limited applicability for mobile application testing, etc. Because of these challenges CompanyX had in several projects abandoned VGT in favor of a 2nd generation approach referred to as the Test interface. The Test interface had several beneficial traits, including higher flexibility in CompanyX's context than Sikuli as well as lower maintenance costs. However, the Test interface approach does not verify that the pictorial GUI conforms to the system's requirements and still suffers from the same synchronization problems as other automated test techniques, including VGT.

Further, it was determined that CompanyX's test process is well integrated into the company's organization, which, together with engineering best practices, were instrumental to VGT's successful adoption at the company. Based on a synthesis of these results, combined with results from previous and related work, 14 practitioner oriented guidelines could be defined for the adoption, use and long-term use of VGT in industrial practice.

This study thereby provides an explicit contribution to the body of knowledge of VGT about the long-term industrial use of the technique. Additionally, the study provides a general contribution to the body of knowledge of software engineering that is currently missing studies that focus on the long-term perspective of research solution's use and challenges in industrial practice.

6.8 Appendix A: Interview Questions

Table 6.5 presents the interview questions used during the four interviews at CompanyX.

Phase	#	Interview question
Adoption	1	What was the reason why CompanyX choose to adopt Sikuli?
	2	What did the adoption process look like for the adoption of Sikuli?
	3	What barriers/challenges were observed during the adoption of Sikuli?
	4	How was Sikuli's Java API adopted to fit CompanyX's test process?
	5	Why was Graphwalker chosen as suitable for the test architecture?
	6	What types of tests were performed with Sikuli?
Usage	7	What benefits were observed with Sikuli compared to other types of testing?
	8	What drawbacks were observed with Sikuli compared to other types of testing?
	9	How often were/could the Sikuli scripts executed?
	10	What was used as specifications for the Sikuli scripts?
	11	What types of defects could be identified with Sikuli?
Maint.	12	What was the need for maintenance of the Sikuli scripts?
	13	What was the need to maintain logic contra images in the scripts?
	14	How much tim was required to maintain the test scripts?
	15	What did the maintenance process for the Sikuli scripts look like?
	16	What challenges were identified with maintaining the Sikuli scripts?
	17	What was the main cause why the Sikuli scripts required maintenance?
Aband.	18	What caused the abandonment of Sikuli for the Test Interface?
	19	What was the timespan from adoption to abandonment of Sikuli?
	20	Has the abandonment of Sikuli caused any new challenges for CompanyX?

Table 6.5: *Interview protocol used during the interviews at CompanyX.* ***Maint.*** *- Maintenance,* ***Aband.*** *- Abandonment.*

Chapter 7

Paper F: VGT-GUITAR

Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: An Empirical Study

E. Alégroth, G. Zebao, R. Oliveira, A. Memon

Accepted at the 8th International Conference on Software Testing Verification and Validation (ICST'2015), Graz, April 13-17, 2015.

Abstract

In this paper we present the results of a two-phase empirical study where we evaluate and compare the applicability of automated component-based Graphical User Interface (GUI) testing and Visual GUI Testing (VGT) in the tools GUITAR and a prototype tool we refer to as VGT GUITAR. First, GUI mutation operators are defined to create 18 faulty versions of an application on which both tools are then applied in an experiment. Results from 456 test case executions in each tool show, with statistical significance, that the component-based approach reports more false negatives than VGT for acceptance tests but that the VGT approach reports more false positives for system tests. Second, a case study is performed with larger open source applications, ranging from 8,803-55,006 lines of code. Results show that GUITAR is applicable in practice but has some challenges related to GUI component states. The results also show that VGT GUITAR is currently not applicable in practice and therefore requires further research and development.

Based on the study's results we present areas of future work for both test approaches and conclude that the approaches have different benefits and drawbacks. The component-based approach is robust and executes tests faster than the VGT approach, with a factor of 3. However, the VGT approach can perform visual assertions and is perceived more flexible than the component-based approach. These conclusions let us hypothesize that a combination of the two approaches is the most suitable in practice and therefore warrants future research.

7.1 Introduction

Today's Graphical User Interface (GUI) applications are complex, run on multiple devices (with different output capabilities), are multi-language, and use a combination of native (e.g., buttons, menus) and advanced (custom) widgets. The modes of interaction with these applications are also evolving, from button clicks to gestures. All of these advances in technology combined with the need for flexibility create challenges for GUI testing (or system testing via the GUI). The oldest tools (harnesses) are based on screen coordinates from Applications Under Test's (AUT) GUI and they have multiple problems for replay (regression testing), especially when screen resolutions change.

Newer test harnesses use direct widget/component access to instrument and assert GUI correctness through the underlying GUI model based on Widgets, properties of widgets and their values. This approach greatly improves test robustness to AUT change and also facilitates automatic record and replay as well as GUI ripping, test case generation and replay with tools such as GUITAR (a GUI Testing FrAmewoRk) [58]. However, this approach requires GUI component access, e.g. through GUI library access, which limits the usability of these tools to certain programing languages, standard GUI components, non-distributed systems, etc.

An alternative harness type is referred to as Visual GUI Testing (VGT), which uses image recognition of what is shown to the user on the computer monitor for AUT interaction [65, 121]. Image recognition makes VGT applicable to any GUI driven AUT regardless of implementation language, type of components, etc. However, this approach is immature and several challenges have been identified with state-of-practice tools and their image recognition algorithms [50]. Thus, even though the industrial applicability of VGT is shown, evidence for the costs associated with the technique are still limited [64, 131].

Because of these listed advantages and disadvantages, we hypothesize that there are characteristics of applications (containing custom widgets) and testing needs (e.g., multi-platform, multi-language) that lend themselves to a combined component-based and VGT based solution. Characteristics that connect to different types of defects on different layers of AUT abstraction, from the GUI rendered on the screen, to the GUI model, to lower system layers. In this paper, we provide support for the above stated hypothesis from an experiment and a single, holistic, empirical case study [17].

In the experiment, two instances, i.e. tools, of the two test approaches were applied on 18 faulty versions [148] of a GUI driven AUT to evaluate the approaches' fault finding ability and false result frequency. Analysis of the results show, with statistical significance, that the component-based approach reports more false negatives for acceptance tests than the VGT approach, but that the VGT approach reports more false positives during system testing. Supporting the hypothesis that a hybrid tool is the most suitable for different industrial contexts and test purposes.

In the case study, the two tools are applied on three larger, open source, AUTs to evaluate the tools' current applicability in practice. Results show that test cases generated for the component-based approach execute successfully roughly two thirds of the time and the VGT approach, due to the current implementation of the prototype tool, had a success rate of zero percent on

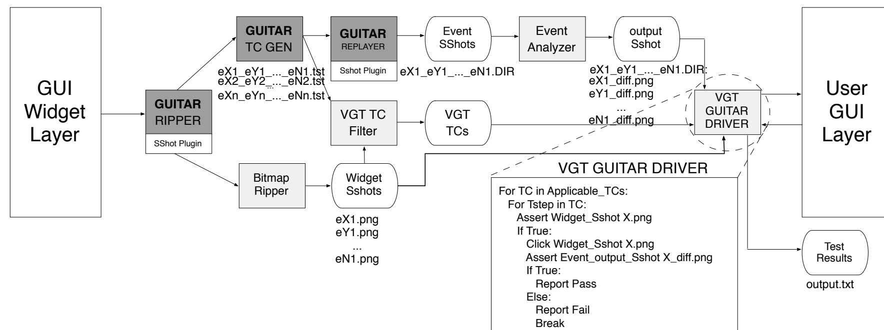

Figure 7.1: *Visualization of VGT GUITAR's architecture.*

the three applications. Based on these results we outline areas of future work for the approach.

The specific contributions of this work are as such:

C1: Comparative, empirical, results regarding the applicability and viability of component-based and VGT based automated testing; and

C2: Support for the need and areas of future work of combining the component-based and VGT approach.

Consequently this study provides a general contribution to the body of knowledge on automated GUI based testing regarding the comparison and unison of the component-based and VGT approaches that currently only has limited support [131]. Further, the study provides an explicit contribution to the body of knowledge on VGT regarding model-based VGT test case generation and replay, which has previously only been evaluated based on random test generation [149].

The continuation of this paper will be structured as follows. First, in Section 7.2 we present a background an extended motivation to the necessity and importance of this work. The research methodology for the two case studies is then described in Section 7.3 followed by the research results and analysis in Section 7.4. These results and analysis are then discussed in Section 7.5, followed by related work in Section 7.6 and finally we conclude the paper in Section 7.7.

7.2 Background and Motivation

Market demands on software developers to rapidly produce new software and add new features to existing software create new challenges for software quality assurance. Challenges that have been proposed as solvable with automated testing [7, 31]. However, software is becoming more GUI intensive with innovative means of interaction, functionality and components. Consequently challenging the capabilities of current GUI test harnesses and presents a need for new automated GUI based testing approaches.

Alégroth et al. [64] classify the existing approaches to GUI-based testing in three chronological generations. The first generation uses exact coordinates

that are recorded during manual interaction with the AUT and automatically replayed for regression testing [8,9]. However, this approach is associated with high maintenance costs due to lack of robustness to GUI change, dependence on screen resolution, etc., and has therefore been abandoned.

Second generation GUI based testing operate directly against GUI component properties, which makes the approach more robust to AUT change, has high performance and stable test execution, supports test recording but also GUI ripping and automated test case generation with tools such as GUITAR [58]. However, to access the components, second generation tools require access to the AUT's underlying GUI model, i.e. through GUI libraries. Thereby restricting the tools' applicability to AUTs with known GUI APIs, local systems, etc [64].

Third generation GUI based testing, also referred to as Visual GUI Testing (VGT), uses image recognition in order to interact and assert AUT correctness through the bitmaps rendered on the user's monitor. However, even though the industrial applicability of the technique has been shown [64], there are still gaps in knowledge regarding the approach long-term viability in practice. Furthermore, only pivotal research has been performed on VGT test case generation [149].

These benefits and drawbacks let us hypothesize that there are potential synergy effects between the second and third generation approaches. To the authors' best knowledge, Unified Functional Testing (UFT) is the only currently available tool with this capability [56]. However, UFT does not support automatic test case generation and is therefore associated with development and maintenance costs. Thus, presenting an industrial need for our research that is also supported by related work [67].

This paper presents a comparative study where we evaluate contextual differences that make the component-based approach beneficial to the VGT approach and vice versa. The data was acquired with GUITAR and a prototype tool, referred here to VGT GUITAR, which combines the two test approaches to facilitate automated image recognition based GUI testing, supported by a script engine written in Python and the Sikuli API [54]. VGT GUITAR's architecture is shown in Figure 7.1. The prototype primarily uses an extension of GUITAR's ripper that takes screenshots of the AUT's components complementary to the components' properties. In addition, the tool includes a purely bitmap based ripper that extract component bitmaps from screenshots of the AUT's GUI taken during replay of the GUITAR test cases. Further, the tool relies on GUITAR's test case generator that in turn has been extended with a filter function that removes all test cases that cannot be executed by VGT GUITAR. Filtering is performed by comparing the pool of captured bitmaps from the AUT's GUI against the bitmaps explicitly required to drive each test case. If a test case includes a test step for which no bitmap was identified during ripping, the test case is removed. Filtering is required since the current tool prototype only captures bitmaps for basic Java components, e.g. JButton, JTextField, etc. GUITAR on the other hand can interact with a larger set of components. The filtered test cases are then executed through the VGT script engine that produces test results that for each test step present if the interaction and assertion of the GUI component passed or failed. As such, VGT GUITAR performs an implicit assertion if the component exists on the screen

before it interacts with it, followed by an assertion of the AUT's behavior in the output assertion.

7.3 Methodology

The methodology used in this work was divided into one experiment and one single holistic case study with two units of analysis, i.e. component-based and VGT testing, as described by Runeson and Höst [17]. The experiment was designed to evaluate the two approaches ability to find different types of defects. These defects were seeded into a small, yet representative, application created by the research team. Furthermore, a case study was performed where the two tools were applied on three open source applications to evaluate the tools' current applicability in practice. The following subsections will present the research design in more detail.

7.3.1 Experiment: Fault detection and False results

In order to compare the defect finding ability of the two GUI based test approaches we used the concepts of mutation testing to create 18 GUI level mutant operators. Mutation testing approaches [150] uses mutation operators to change the AUT to create slightly different versions, called mutants. For each mutant operator, shown in Table 7.1, hypotheses were created regarding the expected behavior of GUITAR and VGT GUITAR when applied for regression testing on a mutant created using said operator. Hence, these hypotheses, stated in columns four and five of Table 7.1, represent the test outcome we expected prior to the study based on the tools' individual capabilities. As an example, Mutant Operator 1 specifies the removal of a component from the GUI, which is a common occurrence when software evolves. The hypothesized correct behavior of both tools in this instance is that any test case that interacted with the removed component would fail.

Further, in order to identify if a reported failure was correct, a false positive or a false negative result, we also hypothesized how the mutant operator would impact system and acceptance testing of the AUT. System testing was defined as assertion of the correctness of the AUT's functionality/behavior, i.e. input and output, and acceptance testing was defined as an assertion of correct functionality/behavior also as executed and accepted by a human user. As such, these hypotheses present what a human tester's test result would be if an instance of the mutant operator occurred in the AUT. Note that this infers that the mutation occurred unintentionally since some of these operators could occur during maintenance of the AUT. As an example, for mutant operator one, if a GUI component is not rendered during testing we expect both a system test case of the AUT's behavior as well as an acceptance test case, which also takes the usability, appearance, etc., of the AUT into account, to fail. The hypotheses are stated in the two last columns of Table 7.1.

To test the hypotheses, the research team created a custom Java application, referred to as AppX, with a simple, modifiable, design but with intricate enough functionality to host all the mutants. AppX has an MVC (Model-View-Controller) architecture with two input buttons and two output fields that displayed the output, as shown in Figure 7.2. The top output field was a

textfield that is accessible through component properties but the lower output field is a custom panel that rendered the output with green letters on a black background that can only be accessed visually. This GUI design was chosen to make the comparison of the two approaches' capabilities fair by ensuring that both techniques could assert the correctness of the AUT. Hence, GUITAR can perform assertions through the top output field's GUI component data and VGT can visually asserts the rendered output in both the top and lower output fields. The rationale behind using AppX rather than an open source AUT for the experiment is to limit confiding factors and to have control of the mutant creation. We also claim that the external validity of the acquired results are high, despite the AUT's lesser size, since the compared test approaches operate on the highest levels of system abstraction which effectively transforms the AUT's business logic into a black box that we stimulate with GUI input to acquire specific GUI output. As such, the size and complexity of the AUT's business logic is irrelevant for the comparison of the two approaches ability to find GUI level regression defects.

To provide further support for our claim about the external validity the reader is referred to Figure 7.3. In the figure, box size represent the depth of business logic in an AUT. At the top of the figure, the business logic of an application with a deterministic implementation of the traveling salesman problem is shown. The theoretical application takes a list of graph nodes, Xs, as input through the GUI and then displays an ordered list, Ys, with the nodes sorted the way they should be optimally traversed. Since the algorithm is deterministic it will always return the same ordered list, Ys, for a given input Xs. In the bottom of the figure we visualize the logic of an AUT where the traveling salesman algorithm has been mocked to a single static method that takes the same list, Xs, as input and returns the correct ordered list Ys as output without any computation. From a user's perspective the behavior of the two applications, on a GUI level, are equivalent, which infers that simplified applications can be used to simulate how GUI level tests behave also on applications with advanced business logic. If the application in the bottom of Figure 7.3 is changed/mutated such that it displays another list Zs, where Zs $\neq$ Ys, it would appear to the user as a defect in the business logic in the application at the top of the figure. Hence, as a defect in the logic that could have been introduced, for instance, during maintenance of the application, i.e. a regression defect. As such, the results acquired during the experiment with AppX are perceived to have the same external validity as results acquired for GUI-based regression testing of a larger application with similar types of GUI components.

Another rationale for the use of AppX was to remove confounding factors from the experimental results caused by defects or other uncontrollable implementation related aspects of another application. By keeping AppX small we could therefore be more certain that the results we acquired were caused by manipulation of the control variables. An alternative had been to perform the experiment on several different applications and through randomization remove the impact of implementation specific problems, which is therefore a potential subject of future work to verify our results.

The mutant operators in Table 7.1 were manually seeded to create 18 different versions of AppX. For instance, for mutation operator 1 the increment

#	Type	Mutant op.	VGT hyp.	Comp. hyp.	Sys. test	Acc. test
1	Rem.	Remove completely	F	F	F	F
2	Rem.	Invisible	F	F	F	F
3	Rem.	Remove listener	F	P	F	F
4	Dup. / Ins.	Add identical widget	F	P	P	F
5	Dup. / Ins.	Add similar widget	F	P	P	P
6	Dup. / Ins.	Add different widget	P	P	P	P
7	Dup. / Ins.	Add another listener	P	P	F	F
8	Mod.	Expand/Reduce size of windows and widgets will auto-adjust their sizes	F	P	P	P
9	Mod.	Expand size of windows and widgets will not auto-adjust their sizes	P	P	P	P
10	Mod.	Reduce size of windows to hide widgets	F	P	F	F
11	Mod.	Modify location of a widget to a proper location	P	P	P	P
12	Mod.	Modify location of a widget to edges of windows	F	P	F	F
13	Mod.	Modify location of a widget to overlap with another	F	P	F	F
14	Mod.	Modify size of widgets	F	P	P	P
15	Mod.	Modify appearance of widgets	F	P	F	F
16	Mod.	Modify type of widgets (Button changed to TextField)	F	F	F	F
17	Mod.	Modify GUI library for widgets (Swing button changed to AWT Button)	P	F	P	P
18	Mod.	Expand/Reduce size of windows and widgets will auto-adjust their sizes	F	P	P	P

Table 7.1: *Mutant operators and hypotheses of the results of applying each of the operators on App X.* ***op.*** *- operator,* ***hyp.*** *- hypothesis,* ***Sys.*** *- System,* ***Acc.*** *- Acceptance.*

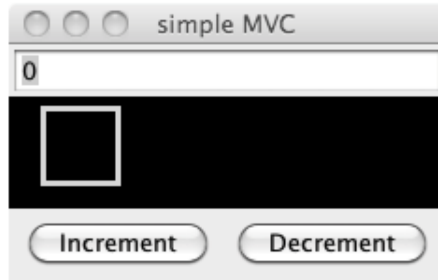

Figure 7.2: *Original GUI for AppX.*

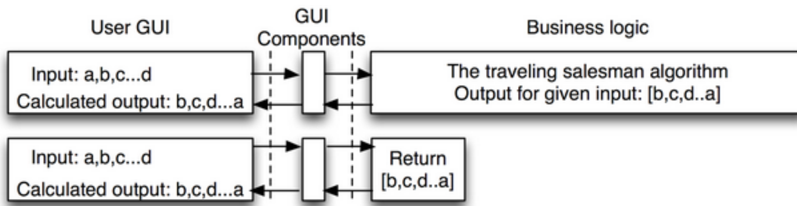

Figure 7.3: *Visualization of two applications with different business logic complexity but with the same behavior for a given input.*

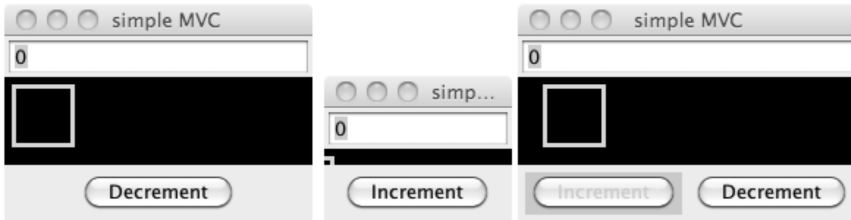

Figure 7.4: *To the left, the result of applying the first mutation operator on AppX, i.e. removing the increment button. In the middle, the result of applying the 10th mutation operator, i.e. reducing the size of the application window to hide widgets. To the right, the 15th mutation operator, i.e. modification of the appearance of widgets.*

button was removed from the view class and the listener from the control class. Figure 7.4 shows three mutated instances of AppX, to be compared to the original GUI in Figure 7.2. The seeding process was performed systematically for each operator by changing as few lines of code as possible for the instance to manifest the mutant. Each instance of AppX was then tested manually to ensure that the introduced mutant had not affected other aspects of the AUT's functionality.

Once the 18 instances had been created, a complete/holistic test suite was created for the original version of AppX that was executed three times on each instance to evaluate the average number of identified defects, false positives, false negatives and test case execution time. The test result was classified as correct if the test case terminated after completing the test case or if it

terminated correctly after it had identified a mutant. The test result was classified as a false positive if the test case failed when it was expected to pass and as a false negative if it passed but was expected to fail. Categorization of the test results was performed through manual inspection of the test results in comparison to the expected behavior as stated in Table T1. As such, eight sets of results were acquired for:

1. False positive component-based system tests;
2. False positive VGT-based system tests;
3. False negative component-based system tests;
4. False negatives VGT-based system tests;
5. False positive component-based acceptance tests;
6. False positive VGT-based acceptance tests;
7. False negative component-based acceptance tests; and
8. False negative VGT-based acceptance tests.

The stated hypotheses were considered supported if all applicable test cases of the test suite conformed to the expected, predefined, test behavior, e.g. if the hypothesis was that GUITAR would report failed test cases for a certain mutation, all test cases affected by the mutation had to fail in order for the hypothesis to be considered supported. Note that in cases where the mutation was only applied to one GUI component of the AUT, test cases that did not interact with said component were assumed to pass as in the original version of AppX.

The results of the experiment were then analyzed using formal statistics with the non-parametric Wilcoxon test to identify, with statistical significance, if there was any difference between the two test approaches' test results. The non-parametric test was used because normality analysis with the Andersson-Darling normality test showed that none of the sets were normally distributed.

7.3.2 Case study: Applicability in practice

As a second part of our evaluation of the two approaches, a case study was performed with the purpose to evaluate the component-based and VGT-based approaches' applicability in practice. Due to VGT GUITAR's dependence on the GUITAR ripper and replayer, the case study was performed with three Java applications, which properties have been presented in Table 7.2. These applications were chosen from an existing pool of applications that have been used in previous research to evaluate GUITAR's applicability [151]. Whilst the size of the applications in lines of code is considered small, compared to applications in industrial practice, the applications have rich GUI's with many different components and events that, according to the reasoning presented in Section 7.3.1, provide the results with a high degree of external validity also for industrial grade applications.

The case study was categorized as a single holistic case study with two units of analysis, being the component-based and VGT approaches represented

Application	Version	LOC	# of Windows	# of Events
Rachota	3.4.0.8	8.803	10	149
JEdit	5.1.0	55.006	20	457
JabRef	2.10b2	52.032	49	680

Table 7.2: *Summary of the properties of the chosen open source applications that were used during the case study.*

by GUITAR and VGT GUITAR [17]. We do not classify the study as an experiment because, even though we compare the tool's results descriptively, the purpose was still to explore the individual applicability of the tools.

The study was performed by constructing and executing test suites of 1000 randomly selected test cases for each AUT in GUITAR. These test suites were then filtered, as explained in Section 7.2, to create test suites executable with VGT GUITAR. In order to improve the internal validity of the results, the results were captured as the average results over two runs of the the two tools for each of the three applications.

The metrics that were measured during the case study were:

1. The average percentage of VGT executable test cases out of the 1000 generated tests;

2. The average execution time per test case for the two approaches; and

3. The percentage of failed test cases per tool and per application.

A qualitative analysis was then performed of the collected metrics to identify the tools perceived applicability in practice. Additionally, the metrics were compared to test the hypothesis that the applicability of GUITAR is higher than the applicability of VGT GUITAR. Further, the hypothesis that the VGT GUITAR prototype is not currently ready for use in practice was also tested. The rejection criteria of the latter hypothesis were that the number of correctly terminating test cases would be significant in number and that the test execution time would be reasonable.

7.4 Results and Analysis

The following subsections will present the results acquired from the experiment and the case study presented in Section 7.3.

7.4.1 Experiment

The main results from the experiment are summarized in Table 7.3. The table shows the number of false positives and false negatives reported by the component-based approach (GUITAR) and the VGT approach (VGT GUITAR). False results are reported as the average number of false results over three runs of each test suite, i.e. for each version of AppX. As such, a zero in the table indicates that all the test cases in that run terminated correctly,

i.e. either found the defect or did not report a defect when there was none to report. These results were then analyzed using formal statistics where an Adersson-Darling normality test was first applied that showed that none of the samples were normally distributed. Because of the lack of normal distribution a non-parametric Wilcoxon test was chosen to compare the samples at a 95 percent confidence level, results shown in Table 7.4.

The results of the statistical analysis show that there was no statistical significant difference between the two techniques in terms of reported false positives for acceptance tests or false negatives for system tests. However, in regard to false positives, the VGT approach will report statistically significantly more false results for system tests, meaning that the approach will fail test cases that should have been passed. The cause of this result is that the VGT approach requires the GUI components to have similar appearance as the original test suite and be visible on the screen. In contrast, the component-based approach can access components that are hidden or have changed appearance. Thus, because the test case creation is performed by GUITAR, test cases are created that disregard how a human user would interact with the AUT. For instance, for the 10th mutant operator, shown Figure 7.4, the AUT's window was reduced, obscuring the decrement button. GUITAR could identify the button through the GUI model, testing the systems functionality, but VGT failed. As such, if the purpose of the test is to exercise the functionality of the system regardless of its graphical appearance the component-based approach is more suitable.

However, the acquired results also show that the component-based approach reports statistically significantly more false negatives for acceptance tests. Hence, the approach fails to report defective AUT states that a human would report, such as defective AUT behavior, un-interactable GUI, etc. The cause of this result is the same as for the previous stated result, i.e. that the component-based approach can interact with hidden components and that the GUI's actual appearance is omitted from the assertions. Consequently, the VGT approach is more suitable for automated acceptance testing where both the AUT's functionality and appearance need to be taken into account in order to satisfy customer needs.

Special attention should be given to mutated versions 3, 4 and 17 of AppX, presented in Table 7.3. In version 3, one of the GUI component listeners was removed, leaving the GUI component visible, and interactive, but it did not generate any output. The component-based approach, in this case, was not able to identify that no new output was produced because GUITAR was only able to assert the GUI model, i.e. change of GUI component properties, but not the pictorial GUI. Thereby reporting false negative results both in terms of system and acceptance tests. In contrast, due to VGT GUITAR's ability to assert both input and expected output visually, it could detect the faulty AUT behavior.

In version 4 of the AUT an extra increment button was added to the AUT that caused the VGT approach to reported false positive results both for system and acceptance tests. The reason was because of the image recognition algorithm's behavior, which swept the screen from the top left to the bottom right but started from the position where the last match was found. Thus, the tool clicked on the correct increment button half of the time, thereby

providing support to previous work regarding the lack of robustness of the VGT approach [64]. The component-based approach did not have this problem because the properties of the original increment button remained the same.

The third case of interest was for version 17, where the AUT's GUI library was changed from Swing to AWT (Abstract Window Toolkit). Consequently changing the properties of the GUI components but not their appearance. This change caused problems for the component-based approach that reported false positives for all test cases for both system and acceptance tests. In contrast, the VGT approach successfully executed all test cases. Thus providing support for the VGT approach flexibility compared to the component-based approach.

Out of the 18 stated hypotheses, shown in Table 7.1, three results deviated from the expected result. These deviations occurred for mutant operators 4, 5 and 7. In 4, we hypothesized that VGT would fail in all instances when an additional, equivalent, button was added to the GUI. However, as discussed above, only some cases failed due to the image recognition's sweep pattern. In 5 we also hypothesized that the VGT approach would fail, i.e. that adding a similar button would cause problems for the image recognition. However, in this case all test cases behaved accurately. Thus proving the hypothesis wrong. In 7 we hypothesized that VGT would succeed even though an additional listener was added to one of the AUT's buttons. However, because of the listeners impact on the output result, VGT GUITAR's output assertions failed, which from a test point of view was correct behavior but rejected our hypothesis. For all other mutants our hypothesis of expected behavior for both approaches were supported by the results.

456 test cases were executed during the experiment, in each tool, with a total execution time for GUITAR of 591 seconds (on average 4.1, standard deviation 0.2 seconds per test case) and 3321 seconds for VGT GUITAR (on average 23, standard deviation 10.7 seconds per test case). As such the execution time for the VGT approach is significantly higher, which can be explained by the speed of the image recognition and because assertions were performed for both input and output to/from the AUT.

7.4.2 Case study

The collected metrics from the second case study have been summarized in Table 7.5.

In order to get suitable samples, test cases were generated in the order of 100k per application from which the test suites of 1000 test cases were randomly selected. The order of magnitude of generated test cases was controlled by the test case length, which due to state space explosions, therefore were kept between two to four steps per test case depending on application. Trial runs were made to evaluate a suitable number of steps for each application and it was found that, as an example, over three million test cases would have been produced for the JabRef application at length three. Generation of all these test cases was deemed unfeasible with an execution time of several hours that warranted the decision of lower test case length, e.g. length two for JabRef. However, this decision also impacts the test cases representativeness for test cases in practice that are generally based on intricate user scenarios to test specific features. As such, we stress that the results of this study are

#	G. FP ST	V. FP ST	G. FP AT	V. FP AT	G. FN ST	V. FN ST	G. FN AT	V. FN AT	G. CAUSE	V. CAUSE
1	0	0	0	0	0	0	0	0	-	-
2	0	0	0	0	0	0	0	0	-	-
3	0	0	0	0	0	0	7	0	Rendered output	-
4	0	3	0	3	0	0	0	0	-	Several identical widgets
5	0	0	0	0	0	0	0	0	-	-
6	0	0	0	0	0	0	0	0	-	-
7	0	0	0	0	0	0	8	0	Rendered output	-
8	0	8	0	8	0	0	0	0	-	Widget size
9	0	0	0	0	0	0	0	0	-	-
10	0	8	0	0	0	0	8	0	Ignore GUI appearance	Hidden widgets
11	0	0	0	0	0	0	0	0	-	-
12	0	7	0	0	0	0	7	0	Ignore GUI appearance	Hidden widgets
13	0	7	0	0	0	0	7	0	Ignore GUI appearance	Hidden widgets
14	0	7	0	7	0	0	0	0	-	Widget size
15	0	7	0	7	0	0	0	0	-	Widget appearance
16	0	0	0	0	0	0	0	0	-	-
17	8	0	8	0	0	0	0	0	GUI library change	-
18	0	8	0	0	0	0	8	0	Ignore GUI appearance	Widget size change

Table 7.3: *False results for system and acceptance tests from case study 1 with AppX.* ***G.*** *- GUITAR,* ***V.*** *- VGT GUITAR,* ***FP*** *- False positive,* ***FN*** *- False negative,* ***ST*** *- System test,* ***AT*** *- Acceptance test.*

	FP ST	**FP AT**	**FN ST**	**FN AT**
P-Value	0.01236	0.1883	n/a	0.009044
Result	Reject H0	Accept H0	Accept H0	Reject H0
Meaning	VGT reports more false positive system test results	No difference between test approaches for false positive acceptance tests	No difference between test approaches for false negative system tests	GUITAR reports more false negative acceptance test results

Table 7.4: *Results of the statistical analysis of the results presented in Table 7.3 with a non-parametric two-sided Wilcoxon test performed at a 95 % confidence interval.* ***FP*** *- False positive,* ***FN*** *- False negative,* ***ST*** *- System test,* ***AT*** *- Acceptance test.*

indicative of the tools' current applicability in practice but that more work is required to show industrial applicability. Further, we also stress that the state space explosion for GUI level test generation is a problem that requires future research and development in order to raise the applicability of the tools in industrial practice.

The GUITAR test suites for each of the applications were then filtered to extract the test cases that could be executed with the VGT approach, i.e. the VGT GUITAR prototype. Table 7.5 shows that the number of VGT applicable test cases was on average less than five percent of the 1000 generated test cases. This low result was caused by the prototype's current ripper function, implemented in GUITAR, that is unable to adequately capture images of all of the GUI's components. The additional bitmap based ripper function was also found to only add a small percentage of extra images. As such, further development is required to improve upon the bitmap ripping.

After execution of the VGT based test cases, it was found that all of them had failed during execution. A failure was in this context identified as a false positive caused by erroneous test case scenarios, which was supported by results from visual inspection of the test cases. Hence, the test cases required interaction with components accessible to the component-based approach through the GUI model but that were not rendered on the GUI, e.g. menu items that GUITAR can access without opening the menu. Other failures were caused by GUI components being in other states than when they were ripped, e.g. buttons that were enabled during ripping were disabled during replay. The root cause of these problems is that the test generation is performed without taking user interaction with the AUT into consideration, i.e. the test cases lack domain knowledge since they are created from a stateless model that does not ensure event availability at runtime. To verify this claim, test cases were created manually for all three applications, using images ripped by GUITAR and GUITAR's test case template, taking human interaction into consideration. All of the manually created tests passed, on all applications, thereby supporting our previous statement.

Metric	Rachota	JEdit	JabRef
Total # of generated TCs	353954	297568	110164
Test case length	4	3	2
# of applicable TCs (G/VGT)	1000 / 53.5	1000 / 1	1000/ 8.5
Failed TCs applicable in both tools (G/VGT)	4% / 100%	0% / 100%	91.5% / 100%
Failed GUITAR test cases (1000 test cases)	50.1%	16.95%	28.3%
Average exe time (G/VGT)	10.65s / 31.74s	12s / 32s	19.143s / 31.875
Standard dev. exe time (G/VGT)	1.71s / 3.049s	0.8325 / 0	0.377s / 0.353s

Table 7.5: *Summary of the quantitative metrics acquired during the second case study.* ***G*** *- GUITAR,* ***VGT*** *- VGT-GUITAR*

However, shown in Table 7.5, the component-based approach failed on average in 31.78 percent of the test cases generated by GUITAR. Further, GUITAR failed on average 31.83 percent of the time for the test cases also applicable for the VGT approach, i.e. out of the 53.5, 1 and 8.5 test cases respectively for Rachota, JEdit and JabRef. Analysis of the test results for the failing test cases showed that there were three types of failure conditions:

1. Interactions with disabled components;
2. Interactions with null components; and
3. Timeouts.

Root cause analysis of these failures showed that they were primarily dependent on the AUT's initial state during execution. For instance, JEdit is a text editor with the majority of its GUI components related to text manipulation and editing. However, if no document is loaded into the application these components are disabled. GUITAR's ripper could capture the components and create test cases with them but during execution the test cases failed because the components' state. This presents a potential threat to the validity of the study's results since the AUT could have been initiated with an example document that had enabled more GUI components and raised the test success-rate. However, since VGT GUITAR was executed in the same context, with disabled buttons, this choice does not affect the validity of the results of the comparison. Especially since it would be expected that both evaluated tools should be applicable regardless of the AUT's initial state.

Additionally, it was observed that the test execution for the VGT approach was significantly slower (a factor of three) than for the component-based approach. This supports the result from the experiment, where VGT was also

slower, and the cause was once again the implementation of the VGT GUITAR prototype, as explained in Section 7.4.1.

Consequently, our hypothesis that the VGT based GUITAR prototype is not currently applicable in practice is supported. One factor of this conclusion is the immaturity of the prototype but also the lack of domain knowledge taken into account during test case generation, which also affects the component-based approach, as shown by the high failure rate for disabled components. This result also shows the importance of placing the AUT in a suitable state before ripping and executing the test cases. In addition it presents a need, and area of future work, for smarter test suite selection algorithms to mitigate false results due to component state and pictorial access to the component. Specifically, this functionality is required for a hybrid tool since ripping with the component-based approach will result in test cases that instrument components visible in the GUI model but not necessarily on the rendered GUI, e.g. menu items or components outside the area of view.

Thus, we state that our hypothesis that GUITAR is significantly more applicable in practice than the VGT GUITAR prototype is supported. Especially if the AUT's initial state is considered such that disabled GUI components are not present, etc. However, it must be noted that whilst the VGT prototype performs both input and output assertions of the AUT's behavior, GUITAR only asserts that input can be performed and that the AUT does not report an exception. GUITAR can also assert AUT state files but these assertions are still limited since they do not evaluate that the rendered GUI state actually conforms to the state file model of the GUI. Thus providing further support for the need of a tool that combines the component-based and VGT approaches to facilitate both automated system and acceptance testing.

7.5 Discussion

Our study shows that in terms of false test results the component-based approach is more suitable for system testing whilst the VGT approach is more suitable for acceptance testing. We therefore posit that a combination of the two is the most suitable.

Previous work [50] has shown that test execution with VGT suffers from robustness problems related to the approach use of image recognition. However, image recognition provides VGT with flexibility and allows for simple creation of powerful input/output oracles, but as a drawback the image recognition is slow. In contrast, the component-based approach is associated with quick, but also robust, test execution. However, the approach requires that the used tool has access to the AUT, which makes it non-flexible and it restricts its applicability for certain systems, e.g. distributed systems or systems written in several programming languages. Furthermore, component-based oracle creation is a challenge due to the required technical knowledge about the AUT and is despite such knowledge only able to assert the GUI model rather what is actually rendered on the screen.

These properties support our statement regarding the suitability of combining the two approaches since it would allow for robust and fast test execution with flexible oracles that could assert both the GUI model as well as the

graphical output shown to the user. To the authors' best knowledge, Hewlett Packard's (HP) tool Unified Functional Testing (UFT) [56] is the only tool available with this multi-approach functionality. The industrial applicability and how well this functionality supports multi-approach scripts is however unknown.

Another important implication of our results is that component-based tools should not be used for acceptance testing, due to the large number of false negative results. Whilst false positives may increase development cost due to unnecessary root-cause analysis of working code, false negatives can result in lingering defects in the delivered system. Thus stressing the need for industrial practitioners to evaluate the capabilities of their test methods. In addition, this result implies that using only one test method for quality assurance is not enough but rather that testing needs to be performed on several, or perhaps all, levels of system abstraction, from component level to GUI level. This statement is partially supported by Berner et al. [16]. However, further work is required to evaluate the criticality of our statement and we do not suggest that automation is the only solution, e.g. acceptance tests could/should be performed manually to explore the space of potential defects in the AUT. However, due to recent trends in software industry towards more continuous integration/development [152], a need exists that warrants future work regarding fully automated GUI based testing tools. In addition, this need also warrants the development of a completely GUI based version of the GUITAR tool, i.e. a continuation of VGT GUITAR. The reason is because, as discussed in this work, it is not possible to obtain test coverage with the component-based approach for systems where AUT access is restricted, e.g. distributed systems and systems developed in multiple languages.

The developed prototype of VGT GUITAR does not fulfill this need, since, as was seen in the second case study, the prototype is currently not applicable in practice. However, the study does showcase that such a tool could be developed and the experiment also shows its potential of such a tool. Further support is given by the related work into Random Visual GUI Testing [149] in which random inputs and random test case execution was used in combination with image recognition to test a real-world web-application.

7.5.1 Threats to Validity

The first threat to the validity of the presented results is the use of a small application for the experiment that could affect the external validity of the results. However, as discussed in Section 7.3, since GUI level testing abstracts the system's logic into a black-box the results are perceived general for all GUI based software. A greater threat is instead the choice of instances for each of the 18 defined mutation operators, i.e. the instances could have been developed in different ways that perceivably could have affected the results. However, this does not affect the validity of the given set, which do encompass a larger set of real application defects. For instance, for Mutation Operator 1, removal of a GUI component from the GUI, often occurs in practice and also entails cases where the GUI component is changed. In the latter case, GUI change, we have redundant evaluation since the experiment evaluates several types of GUI component change, e.g. changing the appearance, type

and GUI library for the component. As such, the threats to external validity are considered low.

Another threat concerns the case study and the choice of applications. The choice is motivated by the applications previous use in academic research and due to their range of functionality, graphical appearance and means of interaction. These applications are therefore perceived to provide results of sufficient external validity. Furthermore, because of their diversity they add to the internal validity of the results but given that other applications could have been chosen there is a minor threat to the construct validity related to the research design. However, since results from the experiment support the results from the case study this threat is also considered minor, i.e. the overall construct validity of this work is considered adequate.

Third, there is a threat to the external validity of the case study in terms of general applicability. We clearly state that we evaluate the applicability of the tools in practice, which is not to be confused with industrial applicability. Due to the open source nature of the applications and limited data set we make no broader claim of the current state of the approaches. However, for GUITAR, previous research does support the tool's general applicability, whilst for VGT GUITAR the results are very limited and such a subject of future work. Such future work should include also analysis of the costs associated with implementation and general use of the tool, including the cost of analysis and identification of false test results.

Lastly, the results of this work indicate, but do not show, that a combination of the two test approaches would improve GUI test efficiency. However, since this factor was not explicitly evaluated in the study we only claim that the reported results are indicative of such a conclusion. Thus, another important subject of future work warranted by the synergy effects identified and presented in this work.

7.6 Related Work

Mutation testing is a practice where developer mistakes are seeded as intentional faults in a software in order to evaluate the quality of a test technique or tool and thereby show that said tests are adequate to find defects in the system [148, 153]. The technique appeared in the early 70s and has, as shown by Jia and Harman, been applied in many different areas of testing for a plethora of programming languages [148]. However, to the authors' best knowledge, our work is the first where the concepts of mutation testing are applied on a GUI level of system abstraction in order to test GUI-based test cases.

GUITAR is a second generation tool that rips, generates and replays test cases automatically for GUI-based testing [58]. The ripping procedure interacts with the AUT and records events and properties of the GUI components in the AUT that are then used to create an event-flow graph [111]. This graph represents the possible interactions that can be performed on the AUT's GUI and is used to generate test cases, which can be replayed for automated GUI-level regression testing [123,153]. As such, the tool adheres to the model-based testing (MBT) paradigm. Thus, removing the need for costly manual test case development, maintenance and execution. However, due to the tools reliance

on the component-based GUI testing, its applicability is restricted to Java and Python AUTs.

Visual GUI Testing is referred to as third generation GUI based testing and is a tool driven technique, with tools such as Sikuli [54] and JAutomate [67], which uses image recognition for interaction and assertion of an AUT's behavior. The benefits of the technique is its flexibility of use for any GUI based system but because of its immaturity it is also associated with robustness problems, i.e. false test results are reported due to image recognition failure [50].

Previous work has evaluated the benefits and drawbacks of the component-based approach and VGT approach for web-systems [131], but to the authors' best knowledge there is no research that evaluates the false test result rate or potential benefits of combining the two techniques or evaluated their applicability on desktop systems.

7.7 Conclusions

In this work we have performed a comparison of component-based and VGT based GUI level testing through an experiment and a case study. Results from the experiment show, with statistical significance, that the component-based approach reports more false negatives than VGT for acceptance tests but that VGT reports more false positives than the component-based approach for system tests. This result relates to how the approaches interact with the AUT. Whilst the component-based approach interacts and asserts the GUI model that makes it suitable for system testing and inapplicable for acceptance testing. In contrast, VGT interacts with the pictorial GUI that makes the approach sensitive to the GUI's appearance but also applicable for automated acceptance testing.

A case study presented in this work also shows that a tool for automated component-based testing, GUITAR, is applicable in practice, but that the VGT based GUITAR prototype tool still requires future research and development. More explicitly, better test case filtering and bitmap image capture algorithms are required to raise the percentage of VGT applicable test cases. In addition, state space explosions must be mitigated to make it possible to generate test cases representative of industrial grade test cases.

Consequently, this work shows that there are complementary benefits and drawbacks with component-based testing and VGT that infers that a combination of the two approaches would be the most suitable in practice. Future work is therefore warranted into tools that can perform automated GUI based testing with both approaches as well as purely VGT based testing to support the software industry's need for test automation that supports continuous integration, development and deployment.

Chapter 8

Paper G: Failure replication

Replicating Rare Software Failures with Visual GUI Testing: An Industrial Success Story

E. Alégroth, J. Gustafsson, H. Ivarsson, R. Feldt

In submissionl.

Abstract

Not all software defects are found before the software reaches the customer. These defects are reported back to the developer in failure reports that are often ambiguous or incomplete making it impossible to replicate the failure to analyze its root-cause. To make matters worse, some defects are non-frequent or even non-deterministic and only occur after extensive use of a system. In other cases the defects are embedded in old and undocumented code that can not be tested by other means than through manual interaction with the system's graphical user interface (GUI). Combined, these scenarios present a worst case where extensive, costly, manual and tedious work is required. Or is it?

Saab AB is a safety critical air traffic management software developer that faced this worst case scenario when they after several years of failure reports from customers regarding a defect that had been deemed unfeasible to remove was able to find a solution. The solution came through the help of a semi-automated approach based on Visual GUI Testing that allowed the company's testers to replicate the defect in less than 24 hours and remove it within the course of seven days. This article will present the company's success story, their experiences and lessons learned as well as the defect, how it came to be and the approach Saab used to finally remove it.

8.1 Failure replication

Despite rigorous testing, it is common in software development practice that faulty software is delivered to customers [154]. When the system eventually fails, customers complain and send back reports that, to varying degrees, contain descriptions of the failures, logs and other information that may be enough to identify the cause(s) of the failure and to remove the defect(s). However, these reports are often ambiguous or incomplete which requires the failure to be replicated for more systematic study. For failures reproducible with little effort, manual, 'monkey' or exploratory testing [42] can be/are used in practice. However, for infrequent and/or non-deterministic failures, manual efforts are generally too costly. Instead, automated approaches are required, such as automated stress testing [155] and/or long-term testing [156]. However, for legacy systems, where access to technical interfaces (APIs etc.) are either restricted, undocumented or missing, automation becomes a challenge. Simply put, these systems lack the necessary prerequisites for most automated testing tools to be applied.

However, there is one test technique which only prerequisite is that the system under test (SUT) has a GUI. This technique is called Visual GUI Testing (VGT) and is an automated test technique based on tools that use image recognition to interact and assert system conformance through the GUI as shown to the user on the monitor. Interactions are further performed using the operating system's clicks and keyboard types which means that VGT scripts can emulate human user behavior. Whilst previous work into VGT has shown the technique's applicability in practice to automate scenario-based manual test cases there is also work to suggest its applicability for other types of testing, e.g. automated random testing [149].

At Saab, VGT saw another use, which was to provide failure inducing stimuli to replicate an unfrequent failure for root-cause analysis and removal of the underlying defect. The continuation of this article will present how the company achieved this feat.

8.2 Success story acquisition

Saab's success story was collected in situ at Saab during an ongoing research collaboration with the company. VGT had, prior to the herein reported case, been introduced at Saab by the research team but the application of the technique for failure replication was initiated by the company's employees; it was not part of any planned research activities. Consequently the success story resulted from collaboration between industry and academia where academic knowledge was applied to solve a problem in practice. Our collaboration therefore made it natural to jointly write this article. Whilst the academic authors were responsible for the introduction of VGT at Saab the first industrial author was the test lead that initiated the VGT based defect replication and the second industrial author was one of the SUT's original developers.

8.3 Success story at Saab

The company: Saab is a software company developing safety-critical air traffic management (ATM) systems that are delivered to both domestic (Sweden) and international airports. The company has roughly 80 engineers that are spread between two development sites in the swedish cities Gothenburg and Växjö. Their software is developed using a mix of plan-driven and agile development processes that conform to strict european regulations on software quality.

Among the company's many products is a radar data presentation (RDP) system that displays aircraft movement in and around the airport's airspace to the (human) air traffic controllers. The RDP is part of an embedded system that interacts with systems for controlling the airstrip lightning, aircraft flight plans and electronic flight strips. As such, the system has a complex backend with several advanced algorithms, controlled by a, in comparison, relatively thin and simple GUI front-end.

Since the system's conception in 2004, it has been extended with new features on top of the core legacy system. The system is built with several programming languages (although primarily C++), includes many different services, is distributed across several physical computers, etc. Because of these system properties it has become a challenge to test the system automatically, i.e. the system does not fulfill the prerequisites of most automated test frameworks. System verification and validation is instead performed with manual tests, complemented by code reviews and customer driven acceptance tests. Whilst these test activities provide the system with sufficient quality, the company has identified a need for more automation and have gradually worked to increase their knowledge about alternatives as well as evaluated their benefits.

The problem: In 2012, the company started receiving failure reports from two specific customer sites that the delivered RDP system crashed after longer-term use, i.e. three to seven months on average. From the customer failure reports, Saab determined that the defect was within the GUI layer of the software and related to memory resource allocation. However, the reports did not provide enough information to find the actual defect. Additional failure replication and analysis was required but replication turned out to be a challenge due to the perceived months of human input required to trigger the failure. It was therefore deemed unfeasible to identify the defect manually simply due to cost. Automated failure replication was as mentioned also not an option due to the system's legacy.

Instead of resolving the defect, since it manifested so seldom in practice, a compromise was selected: the defect was reported to all customers of the system and was then listed as a known defect. Customers were also advised to reboot the system at intervals to reduce the risk of it crashing. However, this was not a solution and the defect still needed to be resolved, but how?

```
def ChangeMapColor():
    #Change the color scheme of the map
    #using UI controls

def ZoomIn():
```

```
    #Zoom in on the map using the
    #mouse-wheel

#(Additional Feature stimuli methods)

def ChangeMapSettings():
    #Open settings menu and update map
    #settings in pop-up menu

i = 1
while i < 3000: #Arbitrarily set number
    ChangeMapColor()
    ZoomIn()
#(Additional feature stimuli method calls)
    ChangeMapSettings()
```

Listing 8.1: Python pseudo-code example of the VGT based stimulator used to explore and identify the RDP defect.

The solution: The solution to resolve the defect came in 2014, two years after the first failure report and 10 years after the defects conception, through a semi-automated test approach developed by one of Saab's engineers. The approach combined incremental failure analysis from exploratory testing with automated system stimuli provided by Visual GUI Testing with the open-source VGT tool Sikuli [20]. Sikuli is a GUI automation tool in which users write Python-based scripts to emulate user behavior. By extending Sikuli with a thin testing layer on top it can integrate well with other test automation solutions [50,121] for user emulated GUI-based system testing. In Saab's case, a script was developed to stimulate the SUT in order to recreate the failure condition. Simply put, instead of using a human to replicate the failure, at high cost, a script was created to achieve the same thing, at minimal cost.

No assertions were added to the script that instead relied on an implicit oracle [69] triggered when the SUT crashed. The test script consisted of roughly 100 lines of Python code that periodically provided stimuli to different RDP features, as shown in Example 8.1. The features in this instance included changing the settings of the radar presentation, zoom and move the radar map, etc.

The VGT stimulation script was intentionally kept as simple and modular as possible to make it easily changeable. Change was required to facilitate an exploratory practice where incremental changes were made of what features to stimulate in each run of the script. After the first script version had been created it was executed nightly on three reference system installations of the RDP. The script managed to reproduce the SUT failure, which in operational practice had taken several months to manifest, in less than 24 hours. Thus, the script quickly confirmed the existence of a defect but still did not provide enough information about its root cause. The script was therefore changed, by removing features that were not perceived to add to the failure, and re-executed to narrow in on which specific stimuli caused the failure. This practice was performed over and over until the failures root-cause was finally identified. Thus, an approach similar to exploratory testing [42] and the automated test

case “shrinking” process applied in QuickCheck [157].

To make failure analysis more efficient, the test execution was monitored using a screen capture software that recorded the screen while the script was executed. Screen capture has been used previously at Saab for VGT based system testing [50]. The recordings made it possible for Saab’s testers to do post-analysis of the SUT failure, asynchronously to the script execution. This practice was also required for feasibility since the execution time of the script was in the order of hours before the failure occurred. To further support the defect analysis, the Windows task manager was used to monitor memory resource allocation during script execution to determine which, if any, memory resource boundaries were being breached before the failure occurred. Hence, the task manager was kept visible on the screen during test execution and screen recording, making it an integral part of the test practice. Using the task manager rather than a more technical solution to extracting memory resource information made it quicker to add this ‘feature’ of the testing.

The defect: As hypothesized by Saab, the defect was identified to be a memory resource allocation problem in the GUI code of the system. More specifically, the defect was located in custom menu components that included a bitmap of a check sign that was rendered if certain SUT options were activated by the user. Each time the check sign was rendered on the screen it allocated two graphical device interface (GDI) objects, which are Windows API components, which were never deallocated. The memory resource leak was small but built up after many interactions with the SUT and once it reached 10,000 allocated objects, which is the maximum amount of allowed memory resources allocated by a single process, the memory resource leak caused the entire system to crash.

Once the memory resource leak was identified, the defect was quickly resolved but this case shows how seemingly minor and unimportant defects can have dramatic consequences. The reason why the failure had not occurred more often in practice is because the air traffic controllers seldom reconfigure the system’s GUI properties. Therefore, the menus with the check sign are not rendered and the memory resource consumption is kept well under the critical margins. However, since the system was, at the two customer sites that reported the failure, kept running in months on end, the memory resource leak slowly but steadily built up and resulted in system failure.

Post-analysis: Post-analysis of this case showed that it had been possible to identify the memory resource leak manually since the number of GDI objects steadily increased during any interaction with the system, manual or automated, given that the check sign was being rendered. However, to pinpoint that it was a GDI memory resource leak required specific domain and SUT knowledge that was not shared by all the employees at Saab. This knowledge was not shared because the defect was embedded in a legacy component developed by technically skilled people and therefore assumed correct and therefore not tested or code reviewed. Further, due to the developers’ skill level, it was assumed that any produced defects would be within the advanced algorithms and data structures of the system, not caused by simple programming mistakes like the identified defect. Verification, through long-term tests and review practices, were later introduced and are currently performed as part of Saab’s development process. However, long-term tests were executed without

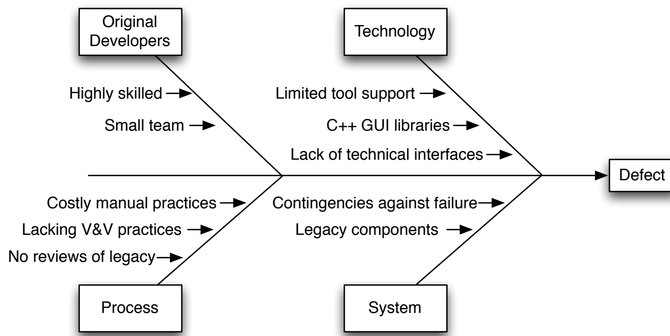

Figure 8.1: *Fishbone diagram summarizing the factors that caused the occurrence and persistence of the defect prior to VGT.*

stimuli which did not trigger the defect and reviews were only performed on new code artifacts.

Figure 8.1 summarizes the factors that lead to the occurrence and persistence of the defect. However, three key reasons why the defect was not managed can be identified from the figure. First, it was deemed too expensive to manually reproduce and identify the defect. Second, the failure caused by the defect was classified as non-safety critical, despite the safety-criticality of the system, because there are contingencies in place at the airports. These contingencies include manual air traffic control processes as well as redundancies within the SUT itself that has several working positions that can operate even if one or several of the working positions go offline. Third, the failure caused the RDP client to crash but not the server backend. As such, by simply rebooting the crashed client it could be brought back into operation.

8.4 Discussion

The presented success story shows that VGT tools can be used to replicate and drive the analysis of non-deterministic and infrequent software failures in systems with restricted access by other test automation tools caused by, for instance, SUT legacy. In addition, this experience shows that test automation can be used to raise system quality and not only as a means to reduce testing costs, which is a common goal for companies adopting more automation.

These results have implications both for industrial practice and for academic researchers. *First*, static long-term tests, where the SUT is left to it's own devices without stimuli for a longer period of time, are not sufficient to find all types of memory resource leaks and SUT misbehavior over time. VGT provides an interface independent and simple solution to provide such stimuli that also has the benefit of mimicking end user interaction. Hence, it can provide GUI level automated stress testing, i.e. subjugating the SUT to harsh inputs with the intention of breaking it [155].

Second, analysis of non-deterministic and unfrequent failures that manifest through the SUT's GUI are no longer out of scope due to unfeasibly high, often because of manual intervention, fault replication costs. Using automated GUI

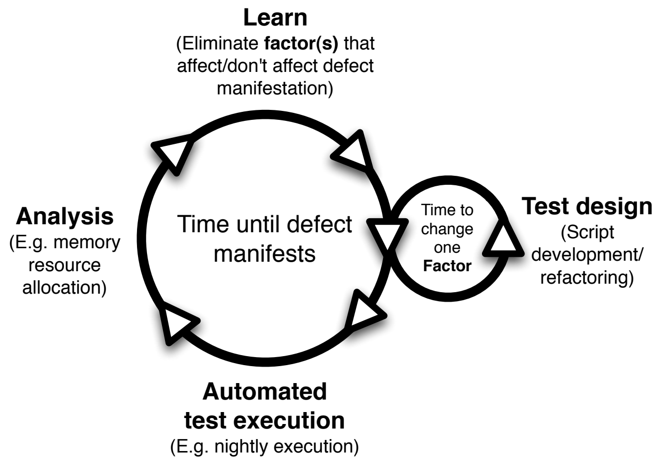

Figure 8.2: *Model of the process used to identify the defect in this case, derived from the exploratory testing process.*

based test techniques for guided system analysis provides a suitable means of identifying these types of defects. Of course, given that the scripts can be modified to include or exclude potential factors, e.g. features and functions, which cause the failure to manifest. It is easy to imagine an even more automated system that based on a simple model of possible interaction events can act as a kind of automated scientist or detective and gradually refine the knowledge about which stimuli are more or less likely to cause the failure.

Third, semi-automated test practices where automated tests are guided by a human oracles can be efficient for replicating failures of defects of unknown origin. The cost of modeling the specification to act as an automated oracle may be prohibitively high or require expertise that companies do not have. In the research literature, automated testing is most often used for regression testing or other test automation where the expected behavior is defined as part of the test case itself. These test cases can be created from specifications or from previously observed defects, e.g. identified during exploratory or other manual testing. However, for latent defects, which only sporadically or infrequently manifests as failures, the time perspective also needs to be considered in addition to feature stimuli. This leads into a hypothesis driven paradigm where root-cause analysis must be driven by iteratively modified automated scripts that explores the input space of the SUT. Consequently following the concepts of exploratory testing [42], simultaneous learning, test design and test execution, but driven by automated stimulation for cost efficiency where manual stimuli is either impractical or infeasible to find the root cause of a failure. A visualization of the adopted exploratory testing process, derived from the success case, has been presented in Figure 8.2.

8.5 Lessons learnt

This case shows that independent of developer skill and complexity of the SUT logic, defects make it into customer releases and can be costly or difficult to manage. The case also shows how legacy software in particular is subject to this challenge for instance due to lack of interfaceability with automated tooling, lack of component knowledge, etc. The success story also shows that VGT can be used to mitigate this problem due to the technique's flexibility, but also that novel test approaches can be created by combining automated test solutions with manual practices. Further, the success story provides empirical evidence from industrial practice, and an actual industrial project, that test automation can be used to identify infrequent and non-deterministic defects. This conclusion is common knowledge but very little explicit empirical support exist in the academic body of knowledge that shows it to be true. Finally the case shows how academic and industrial collaboration can result in effective solutions to industrial problems based on academic knowledge.

The implications of this work are as such that:

1. *VGT can help resolve non-frequent and non-deterministic defects that were previously out of scope due to lack of automation support,*

2. *Automated test tools can be used in novel, semi-automated, scenarios to identify defects,*

3. *Even small defects must be managed since they can have large impacts on a system and the customers perception of its quality,*

4. *Due to feasibility constraints, not all defects can be captured prior to release, implying a need for approaches that can be applied to existing and legacy systems.*

Bibliography

[1] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving GUI-directed test scripts," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on.* IEEE, 2009, pp. 408–418.

[3] ——, "Creating GUI testing tools using accessibility technologies," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on.* IEEE, 2009, pp. 243–250.

[4] M. Finsterwalder, "Automating acceptance tests for GUI applications in an extreme programming environment," in *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering.* Citeseer, 2001, pp. 114–117.

[5] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling manual and automated testing: The autotest experience," in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on.* IEEE, 2007, pp. 261a–261a.

[6] A. Memon, "GUI testing: Pitfalls and process," *IEEE Computer*, vol. 35, no. 8, pp. 87–88, 2002.

[7] E. Dustin, J. Rashka, and J. Paul, *Automated software testing: introduction, management, and performance.* Addison-Wesley Professional, 1999.

[8] A. Onoma, W. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[9] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[10] M. Grechanik, Q. Xie, and C. Fu, "Experimental assessment of manual versus tool-based maintenance of GUI-directed test scripts," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on.* IEEE, 2009, pp. 9–18.

[11] Y. Cheon and G. Leavens, “A simple and practical approach to unit testing: The JML and JUnit way,” *ECOOP 2002Object-Oriented Programming*, pp. 1789–1901, 2006.

[12] E. Sjösten-Andersson and L. Pareto, “Costs and Benefits of Structure-aware Capture/Replay tools,” *SERPS06*, p. 3, 2006.

[13] F. Zaraket, W. Masri, M. Adam, D. Hammoud, R. Hamzeh, R. Farhat, E. Khamissi, and J. Noujaim, “GUICOP: Specification-Based GUI Testing,” in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* IEEE, 2012, pp. 747–751.

[14] M. Olan, “Unit testing: test early, test often,” *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319–328, 2003.

[15] E. Weyuker, “Testing component-based software: A cautionary tale,” *Software, IEEE*, vol. 15, no. 5, pp. 54–59, 1998.

[16] S. Berner, R. Weber, and R. Keller, “Observations and lessons learned from automated testing,” in *Proceedings of the 27th international conference on Software engineering.* ACM, 2005, pp. 571–579.

[17] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[18] R. Potter, *Triggers: GUIding automation with pixels to achieve data access.* University of Maryland, Center for Automation Research, Human/Computer Interaction Laboratory, 1992, pp. 361–382.

[19] L. Zettlemoyer and R. St Amant, “A visual medium for programmatic control of interactive applications,” in *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit.* ACM, 1999, pp. 199–206.

[20] T. Yeh, T. Chang, and R. Miller, “Sikuli: using GUI screenshots for search and automation,” in *Proceedings of the 22nd annual ACM symposium on User interface software and technology.* ACM, 2009, pp. 183–192.

[21] B. A. Kitchenham, T. Dyba, and M. Jorgensen, “Evidence-based software engineering,” in *Proceedings of the 26th international conference on software engineering.* IEEE Computer Society, 2004, pp. 273–281.

[22] I. Sommerville, *Software engineering, 6th ed.* Addison-Wesley Professional, 2000.

[23] M. Huo, J. Verner, L. Zhu, and M. A. Babar, “Software quality and agile methods,” in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International.* IEEE, 2004, pp. 520–525.

[24] J. Highsmith and A. Cockburn, “Agile software development: The business of innovation,” *Computer*, vol. 34, no. 9, pp. 120–127, 2001.

[25] G. Myers, C. Sandler, and T. Badgett, *The art of software testing.* Wiley, 2011.

[26] I. Sommerville, "Software Engineering. International computer science series," 2004.

[27] D. Graham, "Requirements and testing: Seven missing-link myths," *Software, IEEE*, vol. 19, no. 5, pp. 15–17, 2002.

[28] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," *Empirical Software Engineering*, vol. 11, no. 1, pp. 5–31, 2006.

[29] T. Ericson, A. Subotic, and S. Ursing, "TIM - A Test Improvement Model," *Software Testing Verification and Reliability*, vol. 7, no. 4, pp. 229–246, 1997.

[30] T. M. King, A. S. Ganti, and D. Froslie, "Enabling automated integration testing of cloud application services in virtualized environments," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research.* IBM Corp., 2011, pp. 120–132.

[31] C. Lowell and J. Stell-Smith, "Successful Automation of GUI Driven Acceptance Testing," in *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 03)*, Berlin, Heidelberg, 2003, pp. 331–333.

[32] E. Gamma and K. Beck, "JUnit: A cook's tour," *Java Report*, vol. 4, no. 5, pp. 27–38, 1999.

[33] L. Williams, G. Kudrjavets, and N. Nagappan, "On the effectiveness of unit test automation at Microsoft," in *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on.* IEEE, 2009, pp. 81–89.

[34] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *ACM Computing Surveys (CSUR)*, vol. 29, no. 4, pp. 366–427, 1997.

[35] J. Ryser and M. Glinz, "A scenario-based approach to validating and testing software systems using statecharts," in *Proc. 12th International Conference on Software and Systems Engineering and their Applications.* Citeseer, 1999.

[36] R. Binder, *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 2000.

[37] B. Regnell and P. Runeson, "Combining scenario-based requirements with static verification and dynamic testing," in *Proceedings of the Fourth International Workshop on Requirements Engineering-Foundations for Software Quality (REFSQ98), Pisa, Italy.* Citeseer, 1998.

[38] R. Miller and C. Collins, "Acceptance testing," *Proc. XPUniverse*, 2001.

[39] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[40] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. Mantyla, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *Automation of Software Test (AST), 2012 7th International Workshop on*. IEEE, 2012, pp. 36–42.

[41] J. Itkonen, M. V. Mantyla, and C. Lassenius, "Defect detection efficiency: Test case based vs. exploratory testing," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 2007, pp. 61–70.

[42] J. Itkonen and K. Rautiainen, "Exploratory testing: a multiple case study," in *2005 International Symposium on Empirical Software Engineering, 2005*. IEEE, 2005, p. 10.

[43] W. Afzal, A. N. Ghazi, J. Itkonen, R. Torkar, A. Andrews, and K. Bhatti, "An experiment on the effectiveness and efficiency of exploratory testing," *Empirical Software Engineering*, pp. 1–35, 2014.

[44] P. Schipani, "End User Involvement in Exploratory Test Automation for Web Applications," Ph.D. dissertation, TU Delft, Delft University of Technology, 2011.

[45] H. Holmström-Olsson, H. Alahyari, and J. Bosch, "Climbing the" Stairway to Heaven"–A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE, 2012, pp. 392–399.

[46] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 50.

[47] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," *Software Engineering: A Practitioner's Approach*, 2001.

[48] K. Beck and C. Andres, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[49] Z. A. Barmi, A. H. Ebrahimi, and R. Feldt, "Alignment of requirements specification and testing: A systematic mapping study," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 476–485.

[50] E. Alégroth, R. Feldt, and L. Ryrholm, "Visual gui testing in practice: challenges, problemsand limitations," *Empirical Software Engineering*, vol. 20, no. 3, pp. 694–744, 2014.

[51] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon, "Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: an Empirical Study," in *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, Graz, 2015.

[52] E. Horowitz and Z. Singhera, "Graphical user interface testing," *Technical report Us C-C S-93-5*, vol. 4, no. 8, 1993.

[53] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on.* IEEE, 2013, pp. 72–81.

[54] T. Chang, T. Yeh, and R. Miller, "GUI testing using computer vision," in *Proceedings of the 28th international conference on Human factors in computing systems.* ACM, 2010, pp. 1535–1544.

[55] A. Holmes and M. Kellogg, "Automating functional tests using Selenium," in *Agile Conference, 2006.* IEEE, 2006, pp. 6–pp.

[56] T. Lalwani, M. Garg, C. Burmaan, and A. Arora, *UFT/QTP Interview Unplugged: And I Thought I Knew UFT!*, 2nd ed. KnowledgeInbox, 2013.

[57] W.-K. Chen, T.-H. Tsai, and H.-H. Chao, "Integration of specification-based and CR-based approaches for GUI testing," in *Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on*, vol. 1. IEEE, 2005, pp. 967–972.

[58] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: an innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, 2014.

[59] I. K. El-Far and J. A. Whittaker, "Model-Based Software Testing," *Encyclopedia of Software Engineering*, 2001.

[60] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007.* ACM, 2007, pp. 31–36.

[61] P. Fröhlich and J. Link, "Automated test case generation from dynamic models," *ECOOP 2000 Object-Oriented Programming*, pp. 472–491, 2000.

[62] M. Utting and B. Legeard, *Practical model-based testing: a tools approach.* Morgan Kaufmann, 2007.

[63] M. Fowler, *UML distilled: a brief guide to the standard object modeling language.* Addison-Wesley Professional, 2004.

[64] E. Alégroth, "On the Industrial Applicability of Visual GUI Testing," Department of Computer Science and Engineering, Software Engineering (Chalmers), Chalmers University of Technology, Goteborg, Tech. Rep., 2013.

[65] E. Börjesson and R. Feldt, "Automated System Testing using Visual GUI Testing Tools: A Comparative Study in Industry," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* IEEE, 2012, pp. 350–359.

[66] T.-H. Chang, "Using graphical representation of user interfaces as visual references," in *Proceedings of the 24th annual ACM symposium adjunct on User interface software and technology.* ACM, 2011, pp. 27–30.

[67] E. Alégroth, R. Feldt, and H. Olsson, "JAutomate: a Tool for System- and Acceptance-test Automation," *ICST*, 2012.

[68] TestPlant. (2013, Feb.) eggPlant. [Online]. Available: http://www.testplant.com/

[69] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," Technical Report Research Memoranda CS-13-01, Department of Computer Science, University of Sheffield, Tech. Rep., 2013.

[70] R. Harrison and M. Wells, "A meta-analysis of multidisciplinary research," in *Conference on Empirical Assessment in Software Engineering (EASE)*, 2000, pp. 1–15.

[71] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons, 2012.

[72] J. A. Maxwell, *Qualitative research design: An interactive approach.* Sage Publications, Incorporated, 2004.

[73] C. Wohlin, P. Runeson, and M. Höst, *Experimentation in software engineering: an introduction.* Springer Netherlands, 2000.

[74] M. Brydon-Miller, D. Greenwood, and P. Maguire, "Why action research?" *Action research*, vol. 1, no. 1, pp. 9–28, 2003.

[75] C. Robson, *Real world research: a resource for social scientists and practitioner-researchers.* Blackwell Oxford, 2002, vol. 2.

[76] M. Matell and J. Jacoby, "Is there an optimal number of alternatives for Likert scale items? I. Reliability and validity." *Educational and psychological measurement*, 1971.

[77] B. G. Glaser and A. L. Strauss, *The discovery of grounded theory: Strategies for qualitative research.* Transaction Publishers, 2009.

[78] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *Software Engineering, IEEE Transactions on*, vol. 25, no. 4, pp. 557–572, 1999.

[79] R. Barbour and J. Kitzinger, *Developing focus group research: politics, theory and practice.* Sage Publications Limited, 1999.

[80] T. Gorschek, C. Wohlin, P. Carre, and S. Larsson, "A model for technology transfer in practice," *Software, IEEE*, vol. 23, no. 6, pp. 88–95, 2006.

[81] S. Kausar, S. Tariq, S. Riaz, and A. Khanum, "Guidelines for the selection of elicitation techniques," in *Emerging Technologies (ICET), 2010 6th International Conference on.* IEEE, 2010, pp. 265–269.

[82] R. R. Young, "Recommended requirements gathering practices," *CrossTalk*, vol. 15, no. 4, pp. 9–12, 2002.

[83] S. Shiba, "The Steps of KJ: Shiba Method," 1987.

[84] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empirical software engineering*, vol. 10, no. 3, pp. 311–341, 2005.

[85] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, no. 2004, pp. 1–26, 2004.

[86] F. J. Fowler Jr, *Survey research methods.* Sage publications, 2008.

[87] P. Berander and A. Andrews, "Requirements Prioritization," *Engineering and Managing Software Requirements*, 2005.

[88] A. Bowling, *Techniques of questionnaire design.* Open University Press, Maidenhead, UK, 2005.

[89] A. Bryman, "The debate about quantitative and qualitative research: a question of method or epistemology?" *British Journal of Sociology*, pp. 75–92, 1984.

[90] G. Wickström and T. Bendix, "The" Hawthorne effect"what did the original Hawthorne studies actually show?" *Scandinavian journal of work, environment & health*, pp. 363–367, 2000.

[91] V. Kampenes, T. Dybå, J. Hannay, and D. K Sjøberg, "A systematic review of quasi-experiments in software engineering," *Information and Software Technology*, vol. 51, no. 1, pp. 71–82, 2009.

[92] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-experimentation: Design & analysis issues for field settings.* Houghton Mifflin Boston, 1979.

[93] L. Briand, K. El Emam, and S. Morasca, "On the application of measurement theory in software engineering," *Empirical Software Engineering*, vol. 1, no. 1, pp. 61–88, 1996.

[94] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 721–734, 2002.

[95] W. H. Kruskal, "Historical notes on the Wilcoxon unpaired two-sample test," *Journal of the American Statistical Association*, vol. 52, no. 279, pp. 356–360, 1957.

[96] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *IEEE International Conference on Software Engineering (ICSE)*, 2011.

[97] D. Rafi, K. Moses, K. Petersen, and M. Mantyla, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *Automation of Software Test (AST), 2012 7th International Workshop on*, june 2012, pp. 36 –42.

[98] C. Kendall, L. R. Kerr, R. C. Gondim, G. L. Werneck, R. H. M. Macena, M. K. Pontes, L. G. Johnston, K. Sabin, and W. McFarland, "An empirical comparison of respondent-driven sampling, time location sampling, and snowball sampling for behavioral surveillance in men who have sex with men, Fortaleza, Brazil," *AIDS and Behavior*, vol. 12, no. 1, pp. 97–104, 2008.

[99] T. Hellmann, E. Moazzen, A. Sharma, M. Z. Akbar, J. Sillito, F. Maurer *et al.*, "An Exploratory Study of Automated GUI Testing: Goals, Issues, and Best Practices," 2014.

[100] D. Hoffman, "Cost benefits analysis of test automation," *STAR West*, vol. 99, 1999.

[101] P. Li, T. Huynh, M. Reformat, and J. Miller, "A practical approach to testing GUI systems," *Empirical Software Engineering*, vol. 12, no. 4, pp. 331–357, 2007.

[102] P. Hsia, D. Kung, and C. Sell, "Software requirements and acceptance testing," *Annals of software Engineering*, vol. 3, no. 1, pp. 291–317, 1997.

[103] P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen, "Behavior-based acceptance testing of software systems: a formal scenario approach," in *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International.* IEEE, 1994, pp. 293–298.

[104] T. Graves, M. Harrold, J. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[105] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North, "The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends," *Pragmatic Bookshelf*, 2010.

[106] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth, "Automated GUI performance testing," *Software Quality Journal*, pp. 1–39, 2011.

[107] J. Andersson and G. Bache, "The video store revisited yet again: Adventures in GUI acceptance testing," *Extreme Programming and Agile Processes in Software Engineering*, pp. 1–10, 2004.

[108] M. Jovic, A. Adamoli, D. Zaparanuks, and M. Hauswirth, "Automating performance testing of interactive Java applications," in *Proceedings of the 5th Workshop on Automation of Software Test.* ACM, 2010, pp. 8–15.

[109] A. Memon, M. Pollack, and M. Soffa, "Hierarchical GUI test case generation using automated planning," *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 144–155, 2001.

[110] P. Brooks and A. Memon, "Automated GUI testing guided by usage profiles," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* ACM, 2007, pp. 333–342.

[111] A. Memon, "An event-flow model of GUI-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[112] T. Illes, A. Herrmann, B. Paech, and J. Rückert, "Criteria for Software Testing Tool Evaluation. A Task Oriented View," in *Proceedings of the 3rd World Congress for Software Quality*, vol. 2, 2005, pp. 213–222.

[113] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, "Virtual network computing," *Internet Computing, IEEE*, vol. 2, no. 1, pp. 33–38, 1998.

[114] L. Fowler, J. Armarego, and M. Allen, "Case tools: Constructivism and its application to learning and usability of software engineering tools," *Computer Science Education*, vol. 11, no. 3, pp. 261–272, 2001.

[115] S. Eldh, H. Hansson, and S. Punnekkat, "Analysis of Mistakes as a Method to Improve Test Case Design," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on.* IEEE, 2011, pp. 70–79.

[116] J. Itkonen and K. Rautiainen, "Exploratory testing: a multiple case study," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, nov. 2005, p. 10 pp.

[117] J. J. Gutiérrez, M. J. Escalona, M. Mejías, and J. Torres, "Generation of test cases from functional requirements. A survey," in *4ş Workshop on System Testing and Validation*, 2006.

[118] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Software Engineering (ICSE), 2011 33rd International Conference on.* IEEE, 2011, pp. 1066–1071.

[119] D. R. Hackner and A. M. Memon, “Test case generator for GUITAR,” in *Companion of the 30th international conference on Software engineering.* ACM, 2008, pp. 959–960.

[120] V. Vizulis and E. Diebelis, “Self-Testing Approach and Testing Tools,” *Datorzinātne un informācijas tehnoloģijas*, p. 27, 2012.

[121] E. Alégroth, R. Feldt, and H. Olsson, “Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study,” *ICST*, 2012.

[122] A. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping: Reverse engineering of graphical user interfaces for testing,” in *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, 2003, pp. 260–269.

[123] A. M. Memon and M. L. Soffa, “Regression testing of GUIs,” in *ACM SIGSOFT Software Engineering Notes*, vol. 28. ACM, 2003, pp. 118–127.

[124] K. Li and M. Wu, *Effective GUI testing automation: Developing an automated GUI testing tool.* Sybex, 2004.

[125] smartbear. (2013, Feb.) TestComplete. [Online]. Available: http://smartbear.com/products/qa-tools/automated-testing-tools

[126] E. Börjesson, “Multi-Perspective Analysis of Software Development: a method and an Industrial Case Study,” *CPL*, 2010.

[127] B. Beizer, *Software testing techniques.* Dreamtech Press, 2002.

[128] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[129] C. Ebert, “The impacts of software product management,” *Journal of Systems and Software*, vol. 80, no. 6, pp. 850–861, 2007.

[130] C. Mongrédien, G. Lachapelle, and M. Cannon, “Testing GPS L5 acquisition and tracking algorithms using a hardware simulator,” in *Proceedings of ION GNSS*, 2006, pp. 2901–2913.

[131] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, “Visual vs. DOM-Based Web Locators: An Empirical Study,” in *Web Engineering*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8541, pp. 322–340.

[132] S. Wagner, “A model and sensitivity analysis of the quality economics of defect-detection techniques,” in *Proceedings of the 2006 international symposium on Software testing and analysis.* ACM, 2006, pp. 73–84.

[133] K. Karhu, T. Repo, O. Taipale, and K. Smolander, “Empirical observations on software testing automation,” in *Software Testing Verification and Validation, 2009. ICST’09. International Conference on.* IEEE, 2009, pp. 201–209.

[134] C. Liu, "Platform-independent and tool-neutral test descriptions for automated software testing," in *Proceedings of the 22nd international conference on Software engineering.* ACM, 2000, pp. 713–715.

[135] M. Fewster and D. Graham, *Software test automation: effective use of test execution tools.* ACM Press/Addison-Wesley Publishing Co., 1999.

[136] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *Reverse Engineering (WCRE), 2013 20th Working Conference on.* IEEE, 2013, pp. 272–281.

[137] A. Kornecki and J. Zalewski, "Certification of software for real-time safety-critical systems: state of the art," *Innovations in Systems and Software Engineering*, vol. 5, no. 2, pp. 149–161, 2009.

[138] A. Höfer and W. F. Tichy, "Status of empirical research in software engineering," in *Empirical Software Engineering Issues. Critical Assessment and Future Directions.* Springer, 2007, pp. 10–19.

[139] E. Börjesson and R. Feldt, "Automated system testing using visual GUI testing tools: A comparative study in industry," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* IEEE, 2012, pp. 350–359.

[140] A. Marchenko, P. Abrahamsson, and T. Ihme, "Long-term effects of test-driven development a case study," in *Agile Processes in Software Engineering and Extreme Programming.* Springer, 2009, pp. 13–22.

[141] H. Kniberg and A. Ivarsson, "Scaling Agile@ Spotify," *online], UCVOF, ucvox. files. wordpress. com/2012/11/113617905-scaling-Agile-spotify-11. pdf*, 2012.

[142] N. Olsson and K. Karl. (2015) Graphwalker: The Open Source Model-Based Testing Tool. [Online]. Available: http://graphwalker.org/index

[143] J. Carver, "The use of grounded theory in empirical software engineering," in *Empirical Software Engineering Issues. Critical Assessment and Future Directions.* Springer, 2007, pp. 42–42.

[144] J. Saldaña, *The coding manual for qualitative researchers.* Sage, 2012, no. 14.

[145] M. Weinstein. (2002) TAMS Analyzer for Macintosh OS X: The native Open source, Macintosh Qualitative Research Tool. [Online]. Available: http://tamsys.sourceforge.net/

[146] C. Wohlin and A. Aurum, "Towards a decision-making structure for selecting a research design in empirical software engineering," *Empirical Software Engineering*, pp. 1–29, 2014.

[147] N. J. Nilsson, *Principles of artificial intelligence.* Tioga Publishing, 1980.

[148] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[149] E. Alégroth, "Random Visual GUI Testing: Proof of Concept," *Proceedings of the 25th International Conference on Software Engineering & Knowledge Engineering (SEKE 2013)*, pp. 178–184, 2013.

[150] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[151] B. N. Nguyen and A. Memon, "An Observe-Model-Exercise* Paradigm to Test Event-Driven Systems with Undetermined Input Spaces," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 216–234, 2014.

[152] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 2006.

[153] P. R. Mateo, M. P. Usaola, and J. Offutt, "Mutation at system and functional levels," in *Proceedings of the 3rd IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2010)*, Paris, France, 2010, pp. 110–119.

[154] L.-O. Damm, L. Lundberg, and C. Wohlin, "Faults slip through a concept for measuring the efficiency of the test process," *Software Process: Improvement and Practice*, vol. 11, no. 1, pp. 47–59, 2006.

[155] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the 2005 conference on Genetic and evolutionary computation.* ACM, 2005, pp. 1021–1028.

[156] V. T. Rokosz, "Long-term testing in a short-term world," *IEEE software*, vol. 20, no. 3, pp. 64–67, 2003.

[157] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq quickcheck," in *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang.* ACM, 2006, pp. 2–10.